aggregate Documentation

Release 0.23.0

Stephen J. Mildenhall

Mar 06, 2024

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Installation

To install from PyPI

```
pip install aggregate
```

See https://pypi.org/project/aggregate/.

1.2 Source Code

The source code is hosted on GitHub, https://github.com/mynl/aggregate.

1.3 Prerequisites

This help assumes you know how to program in Python, understand probability, and are familiar with the concept of an aggregate distribution. Awareness of insurance terminology such as limit, attachment and deductible, and the material covered in SOA exam STAM, CAS exam MAS I, or IFOA CS-2 is helpful.

1.4 License

BSD 3.

1.5 Dependencies

See pyproject.toml. Requirements are split between those needed to run the project, and a larger set needed to build the documentation.

Apart from `sly` and `titlecase`, all run-dependencies are standard.

1.6 Help Parameters and Examples

Warning: All parameters are fabrications. They try to be realistic (or at least not materially unrealistic) but are not intended to be applied to real-world pricing. They are for educational purposes only.

1.7 Help Structure

This help is structured around **access**, **application**, **theory**, and **implementation**. There are six parts.

1. Getting Started (this document).
2. *User Guides*, explaining how to **access** functionality and practical guides explaining how to **apply** it.
3. *API Reference*: all functions, classes, methods, and properties.
4. *Dec Language Reference*: syntax and grammar.
5. *Technical Guides*, covering the underlying **theory** and its specific **implementation**.
6. *Design and Development*, giving some history, the design philosophy, and ideas for future development.

There is also a *Bibliography*.

1.8 Help Coding Conventions

Throughout the help, you will see input code inside code blocks such as:

```
import pandas as pd
pd.DataFrame({'A': [1, 2, 3]})
```

or:

```
In [1]: import pandas as pd

In [2]: pd.DataFrame({'A': [1, 2, 3]})
Out[2]:
   A
0  1
1  2
2  3
```

The first block is a standard Python input, while in the second the `In [1]:` indicates the input is inside a notebook. In Jupyter Notebooks the last line is printed and plots are shown inline.

For example:

```
In [3]: a = 1

In [4]: a
Out[4]: 1
```

is equivalent to:

```
a = 1
print(a)
```

The Python line continuation \ is used to create compact input.

1.9 Numbers and Units

You can choose your own units. The examples include numbers interpreted in ones, thousands, and millions. Amounts are broadly calibrated to make sense in USD, EUR, and GBP.

1.10 `aggregate` Hello World

The only object you need to import to get started is `build`. The quick display function `qd` is a nice-to-have utility function that handles printing with sensible defaults. It is used extensively throughout.

```
In [5]: from aggregate import build, qd

In [6]: build
Out[6]:
underwriter         Rory
version             0.23.0
knowledge           145 programs
update              True
log2                16
debug               False
validation_eps      0.0001
site dir            ~/aggregate/databases
default dir         ~/checkouts/readthedocs.org/user_builds/aggregate/checkouts/
↪latest/aggregate/agg

help
build.knowledge     list of all programs
build.qshow(pat)    show programs matching pattern
build.show(pat)     build and display matching pattern
```

`build` is a `Underwriter` object. It allows you to create all other objects and includes a library of examples, called the knowledge.

Using `build` you can create an `Aggregate` object using an *DecL program*. For example, the program:

```
agg Eg1 dfreq [1:5] dsev [1:3]
```

creates an aggregate distribution called `Eg1`. The frequency distribution is 1, 2, 3, 4, or 5, all equally likely, and the severity is 1, 2, or 3, also equally likely. The mean frequency is 3, the mean severity 2, and hence the aggregate has a mean of 6. It is built and displayed like so:

```
In [7]: a = build('agg Eg1 dfreq [1:5] dsev [1:3]')

In [8]: qd(a)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)    Skew(X) Est Skew(X)
X
Freq     3                       0.4714                    0
Sev      2        2           0 0.40825   0.40825 6.5268e-15  3.2634e-15
Agg      6        6 -9.992e-16 0.52705   0.52705    0.25298     0.25298
log2 = 5, bandwidth = 1, validation: fails sev skew.
```

The DecL program:

```
agg Eg2 5 claims 1000 xs 0 sev lognorm 50 cv 4 poisson
```

creates a realistic insurance portfolio, with 5 expected claims, severity sampled from a 1000 xs 0 layer of a lognormal with mean 50 and CV 4 and Poisson frequency.

`Aggregate` objects act like a discrete probability distribution. There are properties for the mean, standard deviation, coefficient of variation (cv), and skewness.

```
In [9]: a.agg_m, a.agg_sd, a.agg_cv, a.agg_skew
Out[9]: (6.0, 3.1622776601683795, 0.5270462766947299, 0.2529822128134703)
```

They have probability mass, cumulative distribution, survival, and quantile (inverse of distribution) functions.

```
In [10]: a.pmf(6), a.cdf(5), a.sf(6), a.q(a.cdf(6)), a.q(0.5)
Out[10]: (0.102880658436214, 0.4650205761316873, 0.4320987654320987, 6.0, 6.0)
```

It is easy to check some of these calculations. The probability of the minimum outcome of one equals 1/15 (1/5 for a frequency of 1 and 1/3 for a severity of 1) and the maximum outcome of 15 equals 1/1215 (1/5 for a frequency of 5 and (1/3)**5 to draw severity of 3 on each). The object returns the correct values.

```
In [11]: a.pmf(1), 1/15, a.pmf(15), 1/5/3**5, 5*3**5
Out[11]:
(0.06666666666666668,
 0.06666666666666667,
 0.0008230452674897143,
 0.000823045267489712,
 1215)
```

Creating an object automatically adds its specification to the knowledge, with name `Eg1`. Use `build.knowledge` to view the knowledge dataframe.

```
In [12]: qd(build.knowledge.head(), line_width=73, max_colwidth=50, justify='left')

              program                                               \
kind name
agg  A.Dice00  agg A.Dice00 dfreq  [1:6]  dsev  [1]  note{The...
     A.Dice01  agg A.Dice01 dfreq  [1]  dsev  [1:6]  note{Sam...
     A.Dice02  agg A.Dice02 dfreq  [2]  dsev  [1:6]  note{Sum...
     A.Dice03  agg A.Dice03 dfreq  [5]  dsev  [1:6]  note{Sum...
     A.Dice04  agg A.Dice04 dfreq  [1:6]  dsev  [1:6]  note{S...

              spec
kind name
agg  A.Dice00  {'name': 'A.Dice00', 'freq_name': 'empirical',...
     A.Dice01  {'name': 'A.Dice01', 'freq_name': 'empirical',...
     A.Dice02  {'name': 'A.Dice02', 'freq_name': 'empirical',...
     A.Dice03  {'name': 'A.Dice03', 'freq_name': 'empirical',...
     A.Dice04  {'name': 'A.Dice04', 'freq_name': 'empirical',...

In [13]: qd(build.knowledge.query('name == "Eg1"'), line_width=73, max_colwidth=50,
↪ justify='left')

          program                         \
kind name
agg  Eg1   agg Eg1 dfreq  [1:5]  dsev  [1:3]

          spec
kind name
agg  Eg1   {'name': 'Eg1', 'freq_name': 'empirical', 'fre...
```

The *User Guides* contain more details and examples.

CHAPTER TWO

USER GUIDES

The User Guides show how to **access** `aggregate` functionality and **apply** it to solve actuarial problems. It alternates between access-oriented reference guides and problem and application based practice guides. New users should start reading the *Student* or *Actuarial Student* guide and scan through *A Ten Minute Guide to aggregate*. See *Technical Guides* for the theory and implementation details. Sections in the guides marked **Details** can be skipped. There is some duplication between sections to make them independent.

1. *Student* (practice): Introduction to aggregate distributions using simple discrete examples for actuarial science majors and short-term actuarial modeling exam candidates; get started using `aggregate`.
2. *Actuarial Student* (practice): Introduce the `aggregate` library for working with aggregate probability distributions in the context of actuarial society exams (SOA exam STAM, CAS exam MAS I, or IFOA CS-2) and university courses in (short-term) actuarial modeling.
3. *A Ten Minute Guide to aggregate* (reference): A whirlwind introduction—don't expect to understand everything the first time, but you will see what you can achieve with the package. Read in parallel with the *Student* or *Actuarial Student* practice guides. Follows the pandas model, a *long* 10 minutes.
4. *The Dec Language* (reference): Introduce the Dec Language (DecL) used to specify aggregate distributions in familiar insurance terminology.
5. *Individual Risk Pricing* (practice): Applications of the `Aggregate` class to individual risk pricing, including LEVs, ILFs, layering, and the insurance charge and savings (Table L, M), illustrated using problems from CAS Part 8.
6. *Reinsurance Pricing* (practice): Applications of the `Aggregate` class to reinsurance exposure rating, including swings and slides, aggregate stop loss and swing rated programs, illustrated using problems from CAS Parts 8 and 9.
7. *Reserving* (practice, placeholder): Applications of the `Aggregate` class to reserving, including models of loss emergence and determining ranges for IBNR and case reserves.
8. *Catastrophe Modeling* (practice): Applications of the `Aggregate` class to catastrophe risk evaluation and pricing using thick-tailed Poisson Pareto and lognormal models, including occurrence and aggregate PMLs (OEP, AEP) and layer loss costs. Covers material on CAS Parts 8 and 9.
9. *Capital Modeling and Risk Management* (practice, placeholder): Application of the `Portfolio` class to capital modeling, including VaR, TVaR, and risk visualization and quantification. Covers material on CAS Part 9.
10. *Strategy and Portfolio Management* (practice, placeholder): Application of the `Portfolio` and and `Distortion` classes to strategy and portfolio management, including margin (capital) allocation, determining benchmark pricing within a portfolio using alternative pricing methodologies, and the evaluation of reinsurance.
11. *Case Studies* (practice): Using `aggregate` to reproduce the case study exhibits from the book Pricing Insurance Risk and build similar exhibits for your own cases.
12. *Working With Samples* (reference): How to sample from `aggregate` and how to a build a `Portfolio` from a sample. Inducing correlation in a sample using the Iman-Conover algorithm and determining the worst-VaR rearrangement using the rearrangement algorithm.

13. *Published Problems and Examples* (practice): `aggregate` solutions to a wide selection of problems and examples from books (Loss Models, Loss Data Analytics), actuarial exam study notes, and academic papers. Demonstrates the method of solution and verifies the correctness of `aggregate` calculations.

Guides marked **practice** are problem and application based and give possible driving destinations; those marked **reference** are access-based and describe how to unlock the car, start the engine, and engage a gear.

Guides marked **placeholder** are work in progress, often just a sketch of planned content.

2.1 Student

Objectives: Introduction to aggregate distributions using simple discrete examples for actuarial science majors and short-term actuarial modeling exam candidates; get started using `aggregate`.

Audience: New user, with no knowledge of aggregate distributions or insurance terminology.

Prerequisites: Basic probability theory; Python and pandas programming.

See also: *Actuarial Student*.

Contents:

2.1.1 What Is an Aggregate Probability Distribution?

An **aggregate probability distribution** describes the sum of a random number of identically distributed outcome random variables. The distribution of the number called the **frequency** distribution and of the outcome the **severity** distribution.

Examples.

1. Total losses from insurance claims from a portfolio of policies: frequency equals the number of claims and the severity outcome is the amount of each claim.
2. Larvae per unit area (Neyman 1939): frequency is the number of egg clusters per unit area and severity is the number of larvae per egg cluster.
3. Number of vehicle occupants passing a point on the road: frequency is the number of vehicles passing the point and severity is the number of occupants per vehicle.
4. Total transaction value in an exchange: frequency is the number of transactions and severity is the amount of each transaction.

Aggregate distributions are used in many fields and go by different names, including compound distributions, generalized distributions, and stopped-sum distributions.

2.1.2 Formal Construction

Let N be a discrete random variable taking non-negative integer values. Its outcomes give the frequency (number) of events. Let X_i be a series of independent, identically distributed (iid) severity random variables modeling an outcome. An **aggregate distribution** is the distribution of the random sum

$$A = X_1 + \cdots + X_N.$$

N is called the frequency component of the aggregate and X the severity.

An observation from A is realized by:

1. Sample (or simulate) an outcome n from N
2. For $i = 1, \dots, n$, sample X_i
3. Return $A := X_1 + \cdots + X_n$

It is usual to assume that X and N are independent. Check this assumption is reasonable for your use case; it is not always appropriate. For example, consider modeling hourly takings from a shop checkout till as the number of customers served (frequency) and the amount spent by each customer (severity). Larger orders take longer to tabulate and so frequency is negatively correlated with severity. Example 4 above assumes large orders on an exchange are transacted as quickly as small ones.

2.1.3 Simple Example

Frequency N can equal 1, 2, or 3, with probabilities 1/2, 1/4, and 1/4.

Severity X can equal 1, 2, or 4, with probabilities 5/8, 1/4, and 1/8.

Aggregate $A = X_1 + \cdots + X_N$.

Exercise.

1. What are the expected value and CV of N?
2. What are the expected value and CV of X?
3. What are the expected value and CV of A?
4. What possible values can A take? What are the probabilities of each?

Important: Stop and solve the exercise!

The exercise is not difficult, but it requires careful bookkeeping and attention to detail. It would soon become impractical to solve by hand if there were more outcomes for frequency or severity. This is where `aggregate` comes in. It can solve exercise in the following few lines of code, which we now go through step-by-step.

The first line imports `build` and a helper "quick display" function `qd`. You almost always want to start this way.

```
In [1]: from aggregate import build, qd
```

The next three lines specify the aggregate using a Dec Language (DecL) program to describe its frequency and severity components.

```
In [2]: a01 = build('agg Student:01 '
   ...:             'dfreq [1 2 3] [1/2 1/4 1/4] '
   ...:             'dsev [1 2 4] [5/8 1/4 1/8]')
   ...:
```

The DecL program has three parts:

- `agg` is a keyword and `Student:01` is a user-selected name. Names must start with a letter and can include numbers and colons. This clause declares that we are building an aggregate distribution.

- `dfreq` is a keyword to specify the frequency distribution. The next two blocks of numbers are the outcomes `[1 2 3]` and their probabilities `[1/2 1/4 1/4]`. Commas are optional in the lists and only division arithmetic is supported.
- `dsev` is a keyword to specify the a discrete severity distribution. It has the same outcomes-probabilities form as `dfreq`.

The program string is only one line long because Python automatically concatenates strings within parenthesis; it is split up for clarity. It is recommended that DecL programs be split in this way. Note the spaces at the end of each line, see 10 mins formatting.

Use `qd` to print a dataframe of statistics that answer the first three questions: the mean and CV for the frequency (`Freq`), severity (`Sev`) and aggregate (`Agg`) distributions.

```
In [3]: qd(a01)

       E[X]  Est E[X]     Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq   1.75                          0.4738             0.49338
Sev   1.625     1.625            0 0.61056    0.61056    1.5719       1.5719
Agg  2.8438    2.8437 -1.1102e-16 0.66144    0.66144    1.0808       1.0808
log2 = 5, bandwidth = 1, validation: not unreasonable.
```

The columns `E[X]`, `CV(X)`, and `Skew(X)` report the mean, CV, and skewness for each component computed analytically or very accurately with numerical integration. The columns `Est E[X]`, `Est CV(X)`, and `Est Skew(X)` are computed numerically by `aggregate`. For discrete models they equal the analytic answer because the only errors introduced by `aggregate` come from discretizing the severity distribution. That is also why there are no estimates for frequency. `Err E[X]` shows the error (difference, not relative error) in the mean. This handy dataframe can be accessed directly via the property `a01.describe`. The note `log2 = 5, bs = 1` describe the inner workings, discussed in REF.

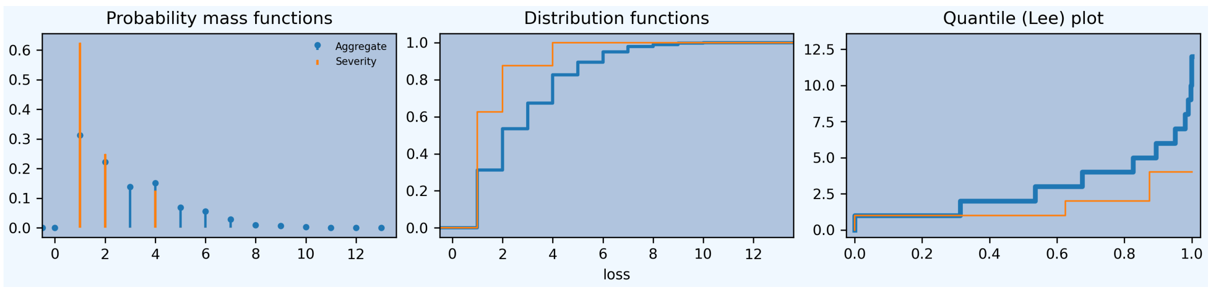
It remains to give the aggregate probability mass function. It is available in the dataframe `a01.density_df`. Here are the probability masses, and distribution and survival functions evaluated for all possible aggregate outcomes.

```
In [4]: qd(a01.density_df.query('p_total > 0')[['p_total', 'F', 'S']])

        p_total       F          S
loss
1.0       0.3125  0.3125     0.6875
2.0      0.22266 0.53516    0.46484
3.0      0.13916 0.67432    0.32568
4.0      0.15137 0.82568    0.17432
5.0     0.068359 0.89404    0.10596
6.0     0.056152  0.9502   0.049805
7.0     0.029297 0.97949   0.020508
8.0    0.0097656 0.98926   0.010742
9.0    0.0073242 0.99658   0.003418
10.0   0.0029297 0.99951 0.00048828
12.0 0.00048828       1          0
```

The possible outcomes range from 1 (frequency 1, outcome 1) to 12 (frequency 3, all outcomes 4). It is easy to check the reported probabilities are correct. It is impossible to obtain an outcome of 11.

For extra credit, here is a plot of the pmf, cdf, and the outcome Lee diagram, showing the severity and aggregate. These are produced automatically by `a01.plot()` from the `density_df` dataframe.

```
In [5]: a01.plot()
```



2.1.4 Exercise - Test Your Understanding

Frequency: 1, 2 or 3 events; 50% chance of 1 event, 25% chance of 2, and 25% chance of 3.

Severity: 1, 2, 4, 8 or 16, each with equal probability.

1. What is the average frequency?
2. What is the average severity?
3. What are the average aggregate?
4. What is the aggregate coefficient of variation?
5. Tabulate the probability of all possible aggregate outcomes.

First, try by hand and then using `aggregate`.

Here is the `aggregate` solution. The probability clause in `dsev` can be omitted when all outcomes are equally likely. The moments and CVs are shown in the table.

```
In [6]: a02 = build('agg Student:02 '
   ...:                'dfreq [1 2 3] [.5 .25 .25] '
   ...:                'dsev [1 2 4 8 16] ')
   ...:

In [7]: qd(a02)

      E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  1.75                        0.4738            0.49338
Sev    6.2       6.2 -1.1102e-16 0.87988   0.87988  0.88905     0.88905
Agg  10.85     10.85 -1.1102e-16 0.81663   0.81663   1.0066      1.0066
log2 = 7, bandwidth = 1, validation: not unreasonable.
```

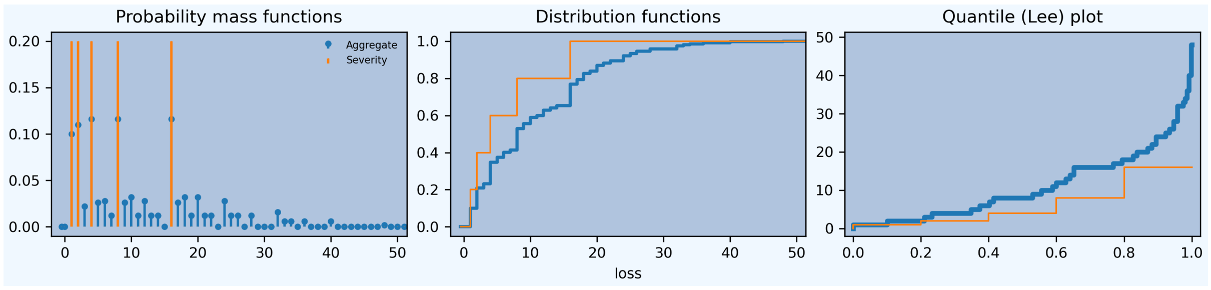
All possible aggregate outcomes are shown next. The largest outcome of 48 has probability 1/4 * (1/5)**3 = 1/500 = 0.002.

```
In [8]: qd(a02.density_df.query('p_total > 0')[['p_total', 'F', 'S']])

      p_total     F            S
loss
1.0       0.1   0.1          0.9
2.0      0.11  0.21         0.79
3.0     0.022 0.232        0.768
4.0     0.116 0.348        0.652
5.0     0.026 0.374        0.626
6.0     0.028 0.402        0.598
7.0     0.012 0.414        0.586
8.0     0.116  0.53         0.47
9.0     0.026 0.556        0.444
10.0    0.032 0.588        0.412
11.0    0.012   0.6          0.4
```

(continues on next page)

(continued from previous page)

```
12.0     0.028 0.628        0.372
...        ...   ...          ...
21.0     0.012 0.882        0.118
22.0     0.012 0.894        0.106
24.0     0.028 0.922        0.078
25.0     0.012 0.934        0.066
26.0     0.012 0.946        0.054
28.0     0.012 0.958        0.042
32.0     0.016 0.974        0.026
33.0     0.006  0.98         0.02
34.0     0.006 0.986        0.014
36.0     0.006 0.992        0.008
40.0     0.006 0.998        0.002
48.0     0.002     1 -2.2204e-16

In [9]: a02.plot()
```



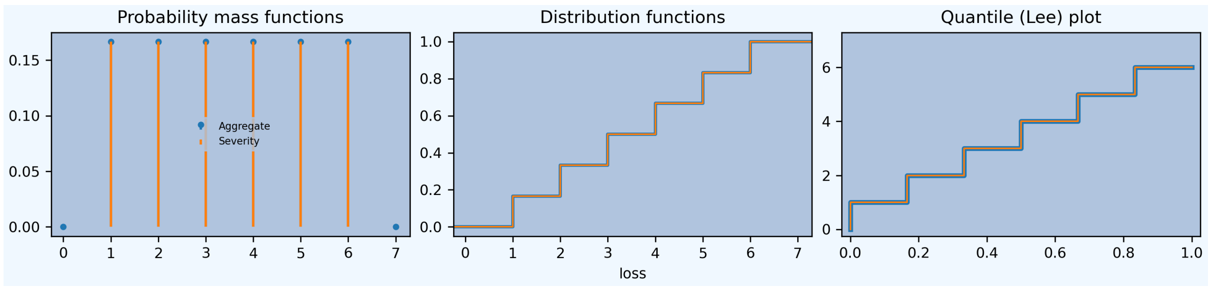
2.1.5 Dice Rolls

This section presents a series of examples involving dice rolls. The early examples are useful because you know the answer and can see `aggregate` is correct.

One Dice Roll

The DecL program for one dice roll.

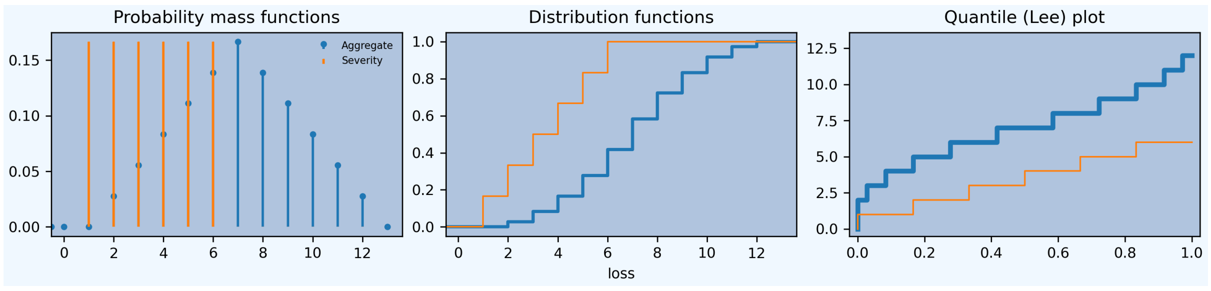
```
In [10]: one_dice = build('agg Student:01Dice '
   ....:                  'dfreq [1] '
   ....:                  'dsev [1:6]')
   ....:

In [11]: one_dice.plot()

In [12]: qd(one_dice)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     1                         0
Sev    3.5      3.5        0 0.48795   0.48795       0   2.8529e-15
Agg    3.5      3.5        0 0.48795   0.48795       0   8.5588e-15
log2 = 4, bandwidth = 1, validation: not unreasonable.
```



Two Dice Rolls

The program for two dice rolls produces a triangular aggregate distribution, as shown in the table and illustrated in the graph (left, probability mass function in blue).

```
In [13]: import numpy as np

In [14]: two_dice = build('agg Student:02Dice '
   ....:                  'dfreq [2] '
   ....:                  'dsev [1:6]')
   ....:

In [15]: two_dice.plot()

In [16]: qd(two_dice)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     2                            0
Sev    3.5      3.5           0 0.48795   0.48795       0  2.8529e-15
Agg      7        7 -3.3307e-16 0.34503   0.34503       0 -4.0346e-14
log2 = 5, bandwidth = 1, validation: not unreasonable.

In [17]: bit = two_dice.density_df.query('p_total > 0')[['p_total', 'F', 'S']]

In [18]: bit['36p'] = np.round(bit.p_total * 36)

In [19]: bit['36p'] = bit['36p'].astype(int)

In [20]: qd(bit)

      p_total        F           S  36p
loss
2.0  0.027778 0.027778     0.97222    1
3.0  0.055556 0.083333     0.91667    2
4.0  0.083333  0.16667     0.83333    3
5.0   0.11111  0.27778     0.72222    4
6.0   0.13889  0.41667     0.58333    5
7.0   0.16667  0.58333     0.41667    6
8.0   0.13889  0.72222     0.27778    5
9.0   0.11111  0.83333     0.16667    4
10.0 0.083333  0.91667    0.083333    3
11.0 0.055556  0.97222    0.027778    2
12.0 0.027778        1 -2.2204e-16    1
```



Twelve Dice Rolls

The aggregate program for twelve dice rolls, which is much harder to compute by hand!

```
In [21]: twelve_dice = build('agg Student:12Dice '
   ....:                     'dfreq [12] '
   ....:                     'dsev [1:6]')
   ....:

In [22]: qd(twelve_dice)

      E[X] Est E[X]    Err E[X]    CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq    12                             0
Sev    3.5      3.5           0 0.48795   0.48795       0   2.8529e-15
Agg     42       42 1.9984e-15 0.14086   0.14086       0   1.4337e-11
log2 = 8, bandwidth = 1, validation: not unreasonable.
```

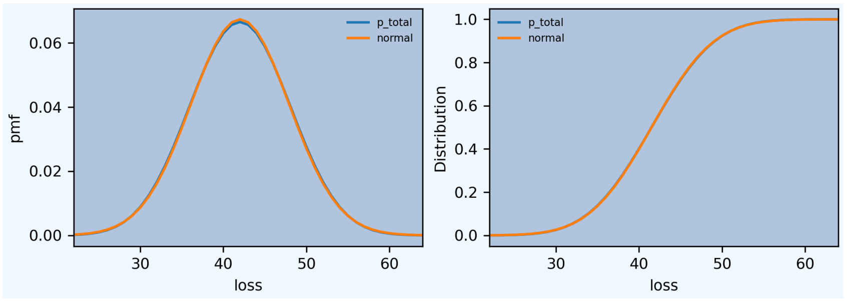
The distribution compared to a moment-matched normal approximation. `fz` is a `scipy.stats` normal random variable created using the `approximate` method. The last two plots show very good convergence to the central limit theorem normal distribution.

```
In [23]: import matplotlib.pyplot as plt

In [24]: fz = twelve_dice.approximate('norm')

In [25]: df = twelve_dice.density_df[['p_total', 'F', 'S']]

In [26]: df['normal'] = np.diff(fz.cdf(df.index + 0.5), prepend=0)

In [27]: qd(df.iloc[32:52])

      p_total        F        S   normal
loss
32.0 0.016609 0.054298   0.9457 0.016196
33.0 0.021737 0.076034  0.92397 0.021233
34.0 0.027592  0.10363  0.89637 0.027054
35.0 0.033997  0.13762  0.86238 0.033502
36.0  0.04069  0.17831  0.82169 0.040322
37.0  0.04733  0.22564  0.77436 0.047165
38.0  0.05353  0.27917  0.72083  0.05362
39.0 0.058887  0.33806  0.66194 0.059245
40.0 0.063026  0.40109  0.59891 0.063621
41.0 0.065643  0.46673  0.53327   0.0664
42.0 0.066539  0.53327  0.46673 0.067353
43.0 0.065643  0.59891  0.40109   0.0664
44.0 0.063026  0.66194  0.33806 0.063621
45.0 0.058887  0.72083  0.27917 0.059245
46.0  0.05353  0.77436  0.22564  0.05362
47.0  0.04733  0.82169  0.17831 0.047165
```

(continues on next page)

(continued from previous page)

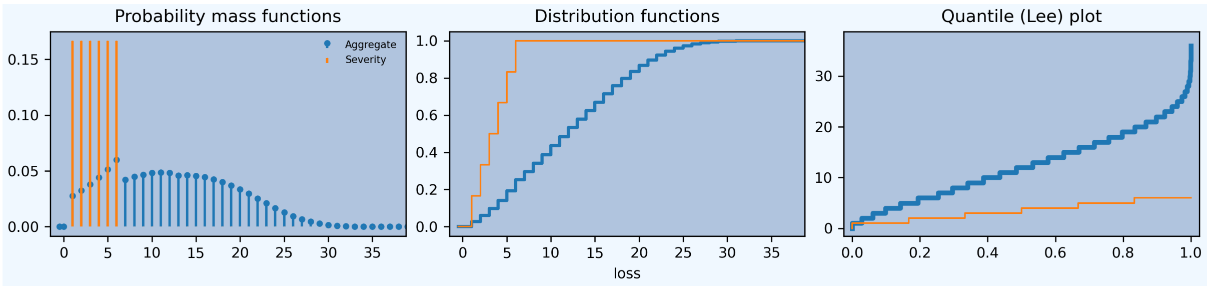
```
48.0  0.04069  0.86238  0.13762 0.040322
49.0 0.033997  0.89637  0.10363 0.033502
50.0 0.027592  0.92397 0.076034 0.027054
51.0 0.021737   0.9457 0.054298 0.021233

In [28]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True); \
   ....: ax0, ax1 = axs.flat; \
   ....: df[['p_total', 'normal']].plot(xlim=[22, 64], ax=ax0); \
   ....: ax0.set(ylabel='pmf'); \
   ....: df[['p_total', 'normal']].cumsum().plot(xlim=[22, 64], ax=ax1);
   ....:

In [29]: ax1.set(ylabel='Distribution');
```



A Dice Roll of Dice Rolls

The last example is a dice roll of dice rolls: throw a dice, then throw that many dice and add up the dots. The result range from 1 (throw 1 first, then 1 again) to 36 (throw 6 first, then 6 for each of the six die).

```
In [30]: dd = build('agg Student:DD '
   ....:            'dfreq [1:6] '
   ....:            'dsev [1:6]')
   ....:

In [31]: qd(dd)

       E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq    3.5                       0.48795                   0
Sev     3.5      3.5           0 0.48795    0.48795         0   2.8529e-15
Agg   12.25    12.25 1.5543e-15 0.55328    0.55328   0.28689      0.28689
log2 = 7, bandwidth = 1, validation: not unreasonable.

In [32]: dd.plot()
```



The largest outcome of 36 has probability 6**-7. See below for a check of the accuracy. Work out the probability of 6 or 7 to better appreciate the work performed by `aggregate`! Why is there a sudden drop between 6 and 7 in the (blue) probability mass function (left hand plot)?

```
In [33]: import pandas as pd

In [34]: a, e = (1/6)**7, dd.density_df.loc[36, 'p_total']

In [35]: pd.DataFrame([a, e, e/a-1],
   ....:     index=['Actual worst', 'Computed worst', 'error'],
   ....:     columns=['value'])
   ....:
Out[35]:
                        value
Actual worst     3.572245e-06
Computed worst   3.572245e-06
error            1.178835e-12
```

We return to this example in *Reinsurance Pricing*.

2.1.6 Summary of Objects Created by DecL

Objects created by `build()` in this guide.

```
In [36]: from aggregate import pprint_ex

In [37]: for n, r in build.qlist('^Student:').iterrows():
   ....:     pprint_ex(r.program, split=20)
   ....:
```

2.2 Actuarial Student

Objectives: Introduce the `aggregate` library for working with aggregate probability distributions in the context of actuarial society exams (SOA exam STAM, CAS exam MAS I, or IFOA CS-2) and university courses in (short-term) actuarial modeling.

Audience: Actuarial science university students and actuarial analysts.

Prerequisites: Familiarity with aggregate probability distributions as covered on actuarial society exams and basic insurance terminology from insurance company operations.

See also: *Student* for a more basic introduction; *User Guides* for other applications.

Contents:

2.2.1 Realistic Insurance Example

Assumptions. You are given the following information about a book of trucking liability insurance business.

1. Premium equals 2000 and the expected loss ratio equals 67.5%.
2. Ground-up severity has been fit to a lognormal distribution with a mean of 100 and CV (coefficient of variation) of 1.75.
3. All policies have a limit of 1000 with no deductible or retention.
4. Frequency is modeled using a Poisson distribution.

You model aggregate losses using the collective risk model.

Questions. Model aggregate losses using the collective risk model and compute the following:

1. The expected insured severity and expected claim count.
2. The aggregate expected value, standard deviation, CV, and skewness.
3. The probability aggregate losses exceed the premium.
4. The probability aggregate losses exceed 2500
5. The expected value of aggregate losses limited to 2500
6. The expected policyholder deficit in excess of 2500

Answers.

Build an aggregate object using simple DecL program. The dataframe `a01.describe` gives the answers to questions 1 and 2. It printed and formatted automatically by `qd(a01)`. Note the validation report in the last line.

```
In [1]: from aggregate import build, qd

In [2]: a01 = build('agg Actuary:01 '
   ...:             '2000 premium at 0.675 lr '
   ...:             '1000 xs 0 '
   ...:             'sev lognorm 100 cv 1.75 '
   ...:             'poisson')
   ...:

In [3]: qd(a01)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 13.948                      0.26776            0.26776
Sev  96.788   96.788  8.8983e-11  1.4339    1.4339   3.5491      3.5491
Agg    1350     1350 -6.8412e-11 0.46809   0.46809  0.88368     0.88368
log2 = 16, bandwidth = 1/8, validation: not unreasonable.
```

The survival function `a01.sf` answers 3 and 4. `qd` is used to print with reasonable defaults. The dataframe `a01.density_df` computes limited expected values (levs) and expected policyholder deficit indexed by loss level, and other values. Querying it answers 5 and 6.

```
In [4]: qd(a01.sf(2000), a01.sf(2500))
0.14971
0.053466

In [5]: qd(a01.density_df.loc[[2500], ['F', 'S', 'lev', 'epd']])

             F        S    lev      epd
loss
2500.0 0.94653 0.053466 1327.7 0.016541
```

2.2.2 College and Exam Questions

College courses and the early actuarial exams often ask purely technical questions. Using assumptions from the *Realistic Insurance Example* answer the following.

1. Compute the severity lognormal parameters mu and sigma.
2. Compute the expected insured severity and expected claim count.
3. Compute the probability the aggregate exceeds the premium using the following matched moment approximations:
    1. Normal
    2. Gamma
    3. Lognormal
    4. Shifted gamma
    5. Shifted lognormal
4. Using the `aggregate` and a lognormal approximation, compute:
    1. The probability losses exceed 2500
    2. The expected value of losses limited to 2500
    3. The expected value of losses in excess of 2500

The code below provides all the answers. `mu_sigma_from_mean_cv` computes the lognormal parameters—one of the most written macro in actuarial science! Start by applying it to the given severity parameters to answer question 1.

```
In [6]: from aggregate import mu_sigma_from_mean_cv

In [7]: import pandas as pd

In [8]: print(mu_sigma_from_mean_cv(50, 1.25))
(3.441531333195883, 0.9700429601128635)
```

The function `a01.approximate` parameterizes all the requested matched moment approximations, returning frozen `scipy.stats` distribution objects that expose `cdf` methods. The `Aggregate` class object `a` also has a `cdf` method. Using these functions, we can assemble a dataframe to answer question 3.

```
In [9]: fz = a01.approximate('all')

In [10]: fz['agg'] = a01

In [11]: df = pd.DataFrame({k: v.sf(2000) for k, v in fz.items()}.items(),
   ....:                columns=['Approximation', 'Value']
   ....:               ).set_index("Approximation")
   ....:

In [12]: df['Error'] = df.Value / df.loc['agg', 'Value'] - 1

In [13]: qd(df.sort_values('Value'))

                Value     Error
Approximation
lognorm       0.13445   -0.1019
slognorm      0.14456  -0.03437
gamma         0.14689 -0.018844
sgamma        0.14745 -0.015088
agg           0.14971         0
norm          0.15183  0.014183
```

The function `lognorm_lev` computes limited expected values for the lognormal. It is used to assemble a dataframe to answer question 4. In this case, the lognormal approximation EPD is over 50% higher than the more accurate estimate provided by `aggregate`.

```
In [14]: from aggregate import lognorm_lev

In [15]: mu, sigma = mu_sigma_from_mean_cv(a01.agg_m, a01.agg_cv)

In [16]: lev = lognorm_lev(mu, sigma, 1, 2500)

In [17]: lev_agg = a01.density_df.loc[2500, 'lev']

In [18]: default = a01.agg_m - lev

In [19]: epd = default / a01.est_m

In [20]: default_agg = a01.est_m - lev_agg

In [21]: bit = pd.DataFrame((lev, default, lev_agg, default_agg, epd, default_agg /
↪ a01.agg_m),
   ....:                  index=pd.Index(['Lognorm LEV', 'Lognorm Default', 'Agg LEV',
   ....:                  'Agg Default', 'Lognorm EPD', 'Agg EPD'],
   ....:                  name='Item'),
   ....:                  columns=['Value'])
   ....:

In [22]: qd(bit)

                    Value
Item
Lognorm LEV        1319.5
Lognorm Default    30.495
Agg LEV            1327.7
Agg Default        22.331
Lognorm EPD      0.022589
Agg EPD          0.016541
```

2.2.3 Advantages of Modeling with Aggregate Distributions

Aggregate distributions provide a powerful modeling paradigm. It separates the analysis of frequency and severity. Different datasets can be used for each. KPW list seven advantages.

1. Only the expected claim count changes with volume. The severity distribution is a characteristic of the line of business.
2. Inflation impacts ground-up severity but not claim count. The situation is more complicated when limits and deductibles apply.
3. Coverage terms impact occurrence limits and deductibles, which affect ground-up severity.
4. The impact on claims frequencies of changing deductibles is better understood.
5. Severity curves can be estimated from homogeneous data. Kaplan-Meier and related methods can adjust for censoring and truncation caused by limits and deductibles.
6. Retained, insured, ceded, and net losses can be modeled consistently.
7. Understanding properties of frequency and severity separately illuminates the shape of the aggregate.

2.2.4 Summary of Objects Created by DecL

Objects created by `build()` in this guide.

```
In [23]: from aggregate import pprint_ex

In [24]: for n, r in build.qlist('^Actuary:').iterrows():
   ....:     pprint_ex(r.program, split=20)
   ....:
```

2.3 A Ten Minute Guide to `aggregate`

Objectives: A whirlwind introduction—don't expect to understand everything the first time, but you will see what you can achieve with the package. Follows the pandas model, a *long* 10 minutes.

Audience: A new user.

Prerequisites: Python programming; aggregate distributions. Read in conjunction with *Student* or *Actuarial Student* practice guides.

See also: *API Reference*, *The Dec Language*.

Contents:

2.3.1 Principle Classes

The `aggregate` package makes working with aggregate probability distributions as straightforward as working with parametric distributions even though their densities rarely have closed-form expressions. It is built around five principal classes.

1. The `Underwriter` class keeps track of everything in its `knowledge` dataframe, interprets Dec Language (DecL, pronounced like deckle, /ˈdɛk(ə)l/) programs, and acts as a helper.
2. The `Severity` class models a size of loss distribution (a severity curve).
3. The `Aggregate` class models a single unit of business, such as a line, business unit, geography, or operating division.
4. The `Distortion` class provides a distortion function, the basis of a spectral risk measure.
5. The `Portfolio` class models multiple units. It extends the functionality in `Aggregate`, adding pricing, calibration, and allocation capabilities.

There is also a `Frequency` class that `Aggregate` derives from, but it is rarely used standalone, and a `Bounds` class for advanced users.

2.3.2 The `Underwriter` Class

The `Underwriter` class is an interface into the computational functionality of `aggregate`. It does two things:

1. Creates objects using the DecL language, and
2. Maintains a library of DecL object specifications called the knowledge. New objects are automatically added to the knowledge.

To get started, import `build`, a pre-configured `Underwriter` and `qd()`, a quick-display function. Import the usual suspects too, for good measure.

```
In [1]: from aggregate import build, qd

In [2]: import pandas as pd, numpy as np, matplotlib.pyplot as plt
```

Printing `build` reports its name, the number of objects in its knowledge, and other information about hyper-parameter default values. `site_dir` is where various outputs will be stored. `default_dir` is for internal package data. The `build` object loads an extensive test suite of DecL programs with over 130 entries.

```
In [3]: build
Out[3]:
underwriter         Rory
version             0.23.0
knowledge           146 programs
update              True
log2                16
debug               False
validation_eps      0.0001
site dir            ~/aggregate/databases
default dir         ~/checkouts/readthedocs.org/user_builds/aggregate/checkouts/
↪latest/aggregate/agg

help
build.knowledge     list of all programs
build.qshow(pat)    show programs matching pattern
build.show(pat)     build and display matching pattern
```

Object Creation Using DecL and `build()`

The Underwriter class interprets DecL programs (*The Dec Language*). These allow severities, aggregates and portfolios to be created using standard insurance language.

For example, to build an `Aggregate` using DecL and report key statistics for frequency, severity, and aggregate, needs just two commands.

```
In [4]: a01 = build('agg TenM:01 100 claims 100 xs 0 sev lognorm 10 cv 1.25 poisson
↪')

In [5]: qd(a01)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   100                          0.1                  0.1
Sev   9.918    9.918  2.8334e-10  1.1639    1.1639   3.4165      3.4165
Agg   991.8    991.8 -4.5175e-10 0.15345   0.15345  0.28923     0.28923
log2 = 16, bandwidth = 1/32, validation: not unreasonable.
```

DecL is supposed to be human-readable, so I hope you can guess the meaning of the DecL code (`TenM:01` is just a label):

```
agg TenM:01 5 claims 1000 xs 0 sev lognorm 50 cv 4 poisson
```

The units are 1000s of USD, EUR, or GBP.

DecL is a custom language, created to describe aggregate distributions. Alternatives are to use positional arguments or key word arguments in function calls. The former are confusing because there are so many. The latter are verbose, because of the need to specify the parameter name. DecL is a concise, expressive, flexible, and powerful alternative.

Important: Formatting a DecL Program

Important: All DecL programs are one line long.

It is best to break a DecL program up to make it more readable. The fact that Python automatically concatenates strings between parenthesis makes this easy. The program above is always entered in the help as:

```
a01 = build('agg TenM:01 '
            '100 claims '
            '100 xs 0 '
            'sev lognorm 10 cv 1.25 '
            'poisson')
```

which Python makes equivalent to:

```
a01 = build('agg TenM:01 100 claims 100 xs 0 sev lognorm 10 cv 1.25 poisson')
```

as originally entered. **Pay attention to spaces at the end of each line!** Entering:

```
a01 = build('agg TenM:01'
            '100 claims'
            '100 xs 0'
            'sev lognorm 10 cv 1.25'
            'poisson')
```

produces:

```
a01 = build('agg TenM:01100 claims100 xs 0sev lognorm 10 cv 1.25poisson')
```

which results in syntax errors.

DecL includes a Python newline `\`. All programs in the help are entered so they can be cut and pasted.

Object Creation from the Knowledge Database

The **knowledge** dataframe is a database of DecL programs and a parsed dictionaries to create objects. `build` loads an extensive library by default. Users can create and load their own databases, allowing them to share common parameters for

- severity (size of loss) curves,
- aggregate distributions (e.g., industry losses in major classes of business, or total catastrophe losses from major perils), and
- portfolios (e.g., an insurer's reference portfolio or educational examples like Bodoff's examples and Pricing Insurance Risk case studies).

It is indexed by object kind (severity, aggregate, portfolio) and name, and accessed as the read-only property `build.knowledge`. Here are the first five rows of the knowledge loaded by `build`.

```
In [6]: qd(build.knowledge.head(), justify="left", max_colwidth=60)

              program                                                          \
kind name
agg  A.Dice00  agg A.Dice00 dfreq  [1:6]  dsev  [1]  note{The roll of a...
     A.Dice01  agg A.Dice01 dfreq  [1]  dsev  [1:6]  note{Same as previ...
     A.Dice02  agg A.Dice02 dfreq  [2]  dsev  [1:6]  note{Sum of the ro...
     A.Dice03  agg A.Dice03 dfreq  [5]  dsev  [1:6]  note{Sum of the ro...
     A.Dice04  agg A.Dice04 dfreq  [1:6]  dsev  [1:6]  note{Sum of a di...

              spec
kind name
agg  A.Dice00  {'name': 'A.Dice00', 'freq_name': 'empirical', 'freq_a':...
     A.Dice01  {'name': 'A.Dice01', 'freq_name': 'empirical', 'freq_a':...
     A.Dice02  {'name': 'A.Dice02', 'freq_name': 'empirical', 'freq_a':...
     A.Dice03  {'name': 'A.Dice03', 'freq_name': 'empirical', 'freq_a':...
     A.Dice04  {'name': 'A.Dice04', 'freq_name': 'empirical', 'freq_a':...
```

A row in the knowledge can be accessed by name using `build`. This example models the roll of a single die.

```
In [7]: print(build['A.Dice00'])
kind                            agg
name                            A.Dice00
spec                            <class 'dict'>
program                         agg A.Dice00 dfreq  [1:6]  dsev  [1]  note{The roll of␣
↪a single dice.}
object                          <class 'NoneType'>
```

The argument `'A.Dice00'` is passed through to the underlying dataframe's `getitem`.

A row in the knowledge can be created as a Python object using:

```
In [8]: aDice = build('A.Dice00')

In [9]: qd(aDice)

      E[X] Est E[X]   Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq   3.5                      0.48795                0
Sev      1        1          0        0          0
Agg    3.5      3.5 4.4409e-16 0.48795   0.48795       0     4.85e-14
log2 = 4, bandwidth = 1, validation: not unreasonable.
```

The argument in this case is passed through to the method `Underwriter.build()`, which first looks for `A.Dice00` in the knowledge. If it fails, it tries to interpret its argument as a DecL program.

The method `build.qlist()` (quick list) searches the knowledge using a regex (regular expression) applied to the names, and returning a dataframe of specifications. `build.qshow()` (quick show) just displays them.

```
In [10]: build.qshow('Dice')

                program                                                        ␣
↪
name                                                                           ␣
↪
A.Dice00          agg A.Dice00 dfreq  [1:6]  dsev  [1]  note{The roll of a single␣
↪dice.}
A.Dice01          agg A.Dice01 dfreq  [1]  dsev  [1:6]  note{Same as previous␣
↪example.}
A.Dice02          agg A.Dice02 dfreq  [2]  dsev  [1:6]  note{Sum of the rolls of␣
↪two dice.}
A.Dice03          agg A.Dice03 dfreq  [5]  dsev  [1:6]  note{Sum of the rolls of␣
```

(continues on next page)

(continued from previous page)

```
↪five dice.}
A.Dice04         agg A.Dice04 dfreq  [1:6]  dsev  [1:6]  note{Sum of a dice roll␣
↪of dice rolls}
A.Dice05         agg A.Dice05 dfreq  [1:4]  dsev  [1:16]  note{Something you can't␣
↪do easily by hand}
```

The method `build.show()` also searches the knowledge using a regex applied to the names, but it creates and plots each match by default. Be careful not to create too many objects! Try running:

```
build.show('Dice')
```

Add argument `return_df=True` to return a list of created objects and a dataframe containing information about each.

`Underwriter` Behind the Scenes

This section should be skipped the first time through.

Each object has a kind property and a name property, and it can be manifest as a DecL program, a dictionary specification, or a Python class instance. The class can be updated or not updated. In detail:

1. kind equals sev for a `Severity`, agg for a `Aggregate`, port for a `Portfolio`, and distortion for a `Distortion` (dist could be distribution);
2. name is assigned to the object by the user; it is different from the Python variable name holding the object;
3. spec is a (derived) dictionary specification;
4. program is the DecL program as a text string; and
5. object is the actual Python object, an instance of a class.

`Underwriter.write()` is a low-level creator function. It takes a DecL program or knowledge item name as input.

- It searches the knowledge for the argument and returns it if it finds one object. It throws an error if the name is not unique. If the name is not in the knowledge it continues.
- It calls `Underwriter.interpret_program()` to pre-process the DecL and then lex and parse it one line at a time.
- It looks up occurrences of `sev.ID`, `agg.ID` (`ID` is an object name) in the knowledge and replaces them with their definitions.
- It calls `Underwriter.factory()` to create any objects and update them if requested.
- It returns a list of `Answer` objects, with kind, name, spec, program, and object attributes.

`Underwriter.write_file()` reads a file and passes it to `Underwriter.write()`. It is a convenience function.

The `Underwriter.build()` method wraps the `Underwriter.write()` and provides sensible defaults to shield the user from its internal details. *build* takes the following steps:

- It calls `write()` with `update=False`.
- It then estimates sensible hyper-parameters and uses them to `update()` the object's discrete distribution. It tries to distinguish discrete output distributions from continuous or mixed ones.
- If the DecL program produces only one output, it strips it out of the answer returned by `write` and returns just that object.
- If the DecL program produces only one portfolio output (but possibly other non-portfolio objects), it returns just that.

`Underwriter.interpret_program()` interprets DecL programs and matches them with the parsed specs in an `Answer(kind, name, spec, program, object=None)` object. It adds the result to the knowledge.

`Underwriter.factory()` takes an `Answer` argument and updates it by creating the relevant object and updating it if `build.update is True`.

A set of methods called `interpreter_xxx()` run DecL programs through parser for debugging purposes, but do not create any output or add anything to the knowledge.

- `Underwriter.interpreter_line()` works on one line.
- `Underwriter.interpreter_file()` works on each line in a file.
- `Underwriter.interpreter_list()` works on each item in a list.
- `Underwriter._interpreter_work()` does the actual parsing.

2.3.3 How `aggregate` Represents Distributions

A distribution is represented as a discrete numerical approximation. To "know or compute a distribution" means that we have a discrete stair-step approximation to the true distribution function that jumps (is supported) only on integer multiples of a fixed bandwidth or bucket size b (called `bs` in the code). The distribution is represented by b and a vector of probabilities $(p_0, p_1, \dots, p_{n-1})$ with the interpretation

$$\Pr(X = kb) = p_k.$$

All subsequent computations assume that **this approximation is the distribution**. For example, moments are estimated using

$$\mathsf{E}[X^r] = b \sum_k k^r p_k.$$

See num how agg reps a dist for more details.

2.3.4 The `Severity` Class

The `Severity` class derives from `scipy.stats.rv_continuous`, see scipy help. It contains a member `stats.rv_continuous` variable `fz` that is the ground-up unlimited severity and it adds support for limits and attachments. For example, the cdf function is coded:

```
def _cdf(self, x, *args):
    if self.conditional:
        return np.where(x >= self.limit, 1,
            np.where(x < 0, 0,
                     (self.fz.cdf(x + self.attachment) -
                     (1 - self.pattach)) / self.pattach))
    else:
        return np.where(x < 0, 0,
            np.where(x == 0, 1 - self.pattach,
                 np.where(x > self.limit, 1,
                     self.fz.cdf(x + self.attachment, *args))))
```

`Severity` can determine its shape parameter from a CV analytically for lognormal, gamma, inverse gamma, and inverse Gaussian distributions, and attempts to use a Newton-Raphson method to determine it for all other one-shape parameter distributions. (The CV is adjusted using the scale factor for zero parameter distributions.) Once the shape is known, it uses scaling to produce the required mean. **Warning:** The numerical methods are not always reliable.

`Severity` computes layer moments analytically for the lognormal, Pareto, and gamma, and uses numerical integration of the quantile function (`isf`) for all other distributions. These estimates can become unreliable for very thick tailed distributions. It uses `self.fz.stats('mvs')` and the object limit to determine if the requested moment actually exists before attempting numerical integration.

`Severity` has a `plot()` method that graphs the density, log density, cdf, and quantile (Lee) functions.

A `Severity` can be created using DecL using any of the following five forms.

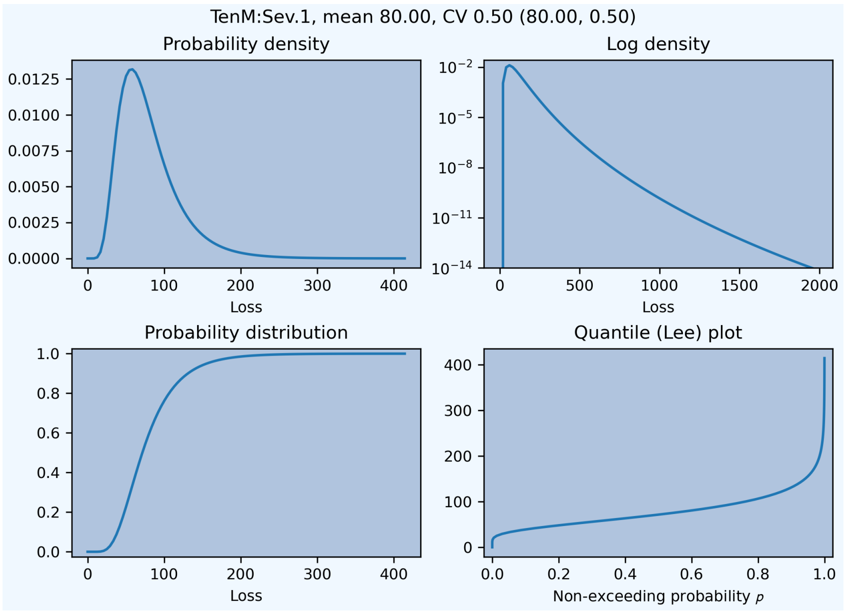
1. `sev NAME sev.BUILDIN_ID` is a knowledge lookup for `BUILTIN_ID`
2. `sev NAME DISTNAME SHAPE1 <SHAPE2>` where `DISTAME` is the name of any `scipy.stats` continuous random variable with zero, one, or two shape parameters, see the DecL/list of distributions.
3. `sev NAME SCALE * DISTNAME SHAPE1 <SHAPE2> + LOC`
4. `sev NAME DISTNAME MEAN cv CV`
5. `sev NAME SCALE * DISTNAME MEAN cv CV + LOC` or `sev NAME SCALE * DISTNAME MEAN cv CV - LOC`

Either or both of `SCALE` and `LOC` can be present. In the mean and CV form, the mean refers to the unshifted, unscaled mean, but the CV refers to the shifted and scaled CV — because you usually want to control the overall CV.

Example.

`lognorm 80 cv 0.5` results in an unshifted lognormal with mean 80 and CV 0.5.

```
In [11]: s0 = build(f'sev TenM:Sev.1 '
   ....:            'lognorm 80 cv .5')
   ....:

In [12]: mf, vf = s0.fz.stats(); m, v = s0.stats()

In [13]: s0.plot(figsize=(2*3.5, 2*2.45+0.15), layout='AB\nCD');

In [14]: plt.gcf().suptitle(f'{s0.name}, mean {m:.2f}, CV {v**.5/m:.2f} ({mf:.2f},
↪{vf**.5/mf:.2f})');

In [15]: print(m,v,mf,vf)
79.99999999999997 1599.999999999991 80.00000000000001 1600.0000000000002
```



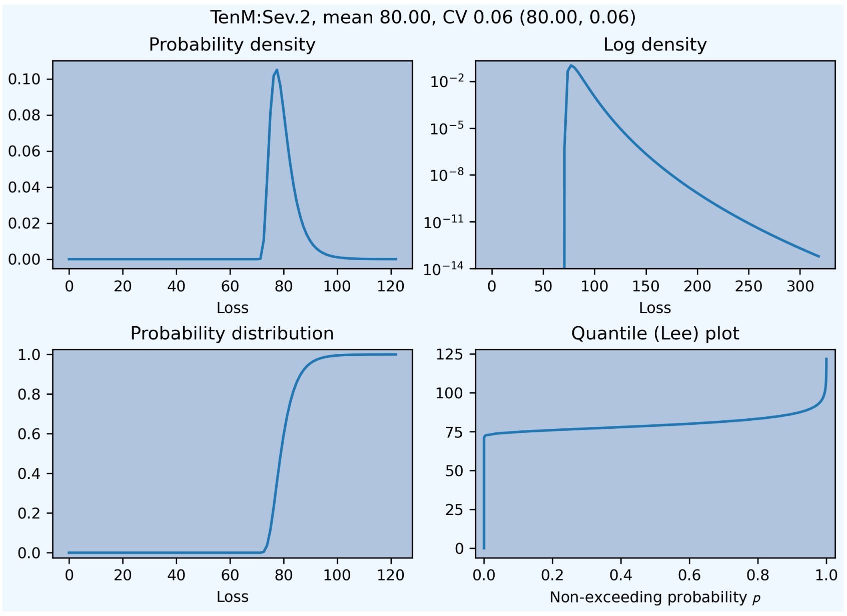
Combining scaling, shifts, and mean/cv entry like so `10 * lognorm 1 cv 0.5 + 70` results in a distribution with mean `10 * 1 + 70 = 80`, a standard deviation of `10 * 0.5 = 5`, and a cv of `5 / 80`.

```
In [16]: s1 = build(f'sev TenM:Sev.2 '
   ....:            '10 * lognorm 1 cv .5 + 70')
   ....:
```

(continues on next page)

(continued from previous page)

```
In [17]: mf, vf = s1.fz.stats(); m, v = s1.stats()

In [18]: s1.plot(figsize=(2*3.5, 2*2.45+0.15), layout='AB\nCD');

In [19]: plt.gcf().suptitle(f'{s1.name}, mean {m:.2f}, CV {v**.5/m:.2f} ({mf:.2f},
↪{vf**.5/mf:.2f})');

In [20]: print(m,v,mf,vf)
80.00000002527862 25.000006201477845 80.0 25.000000000000004
```



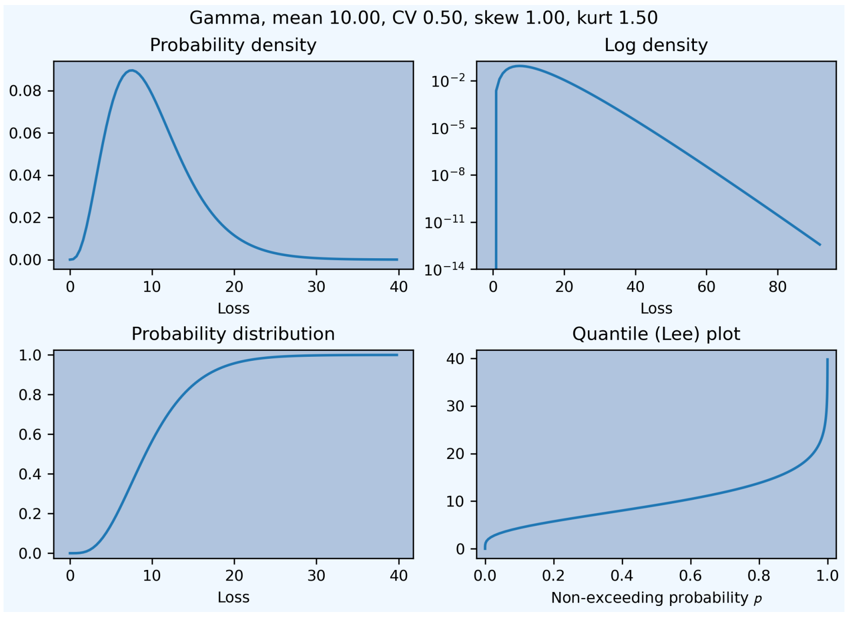
Examples.

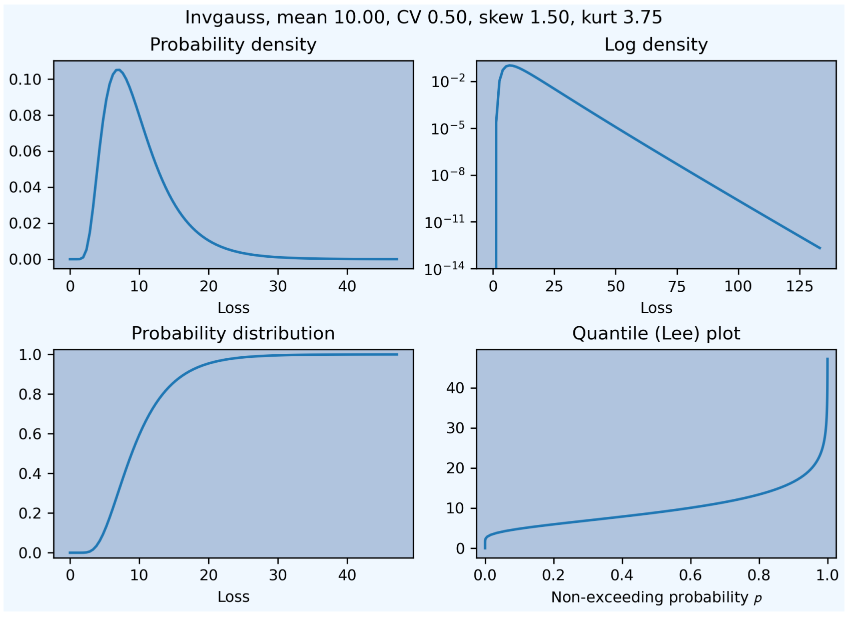
This example compares the shapes of gamma, inverse Gaussian, lognormal, and inverse gamma severities with the same mean and CV. First, a short function to create the examples.

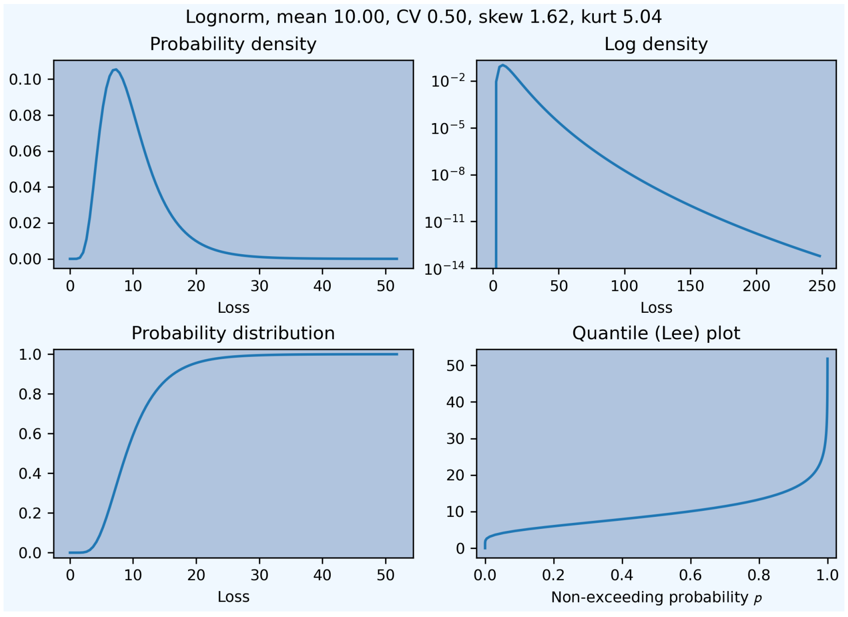
```
In [21]: def plot_example(dist_name):
   ....:     s = build(f'sev TenM:{dist_name.title()} '
   ....:               f'{dist_name} 10 cv .5')
   ....:     m, v, sk, k = s.fz.stats('mvsk')
   ....:     s.plot(figsize=(2*3.5, 2*2.45+0.15), layout='AB\nCD')
   ....:     plt.gcf().suptitle(f'{dist_name.title()}, mean {m:.2f}, '
   ....:                        f'CV {v**.5/m:.2f}, skew {sk:.2f}, kurt {k:.2f}')
   ....:
```

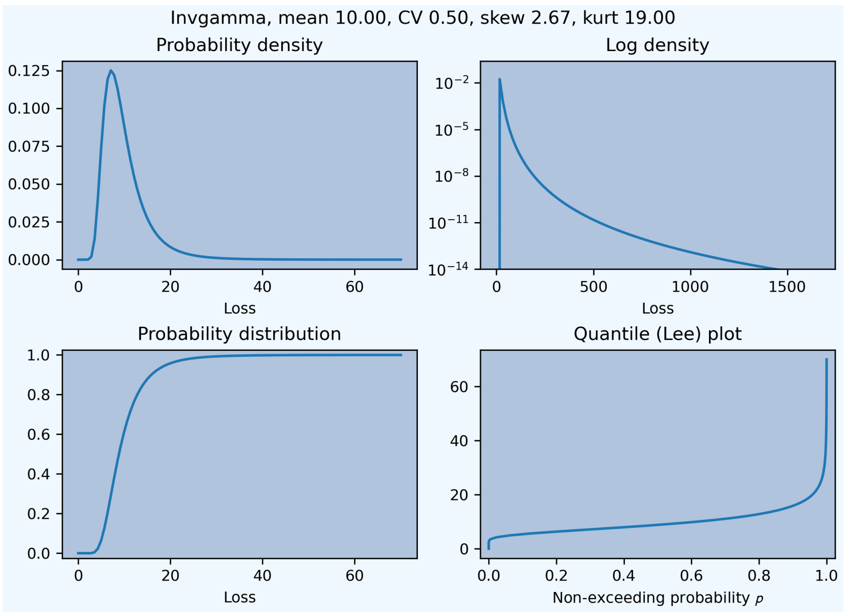
Execute on the desired distributions.

```
In [22]: plot_example('gamma')

In [23]: plot_example('invgauss')

In [24]: plot_example('lognorm')

In [25]: plot_example('invgamma')
```


Gamma, mean 10.00, CV 0.50, skew 1.00, kurt 1.50
Probability density
Log density
Loss
Probability distribution
Quantile (Lee) plot
Non-exceeding probability p


Invgauss, mean 10.00, CV 0.50, skew 1.50, kurt 3.75
Probability density
Log density
Loss
Probability distribution
Quantile (Lee) plot
Non-exceeding probability p

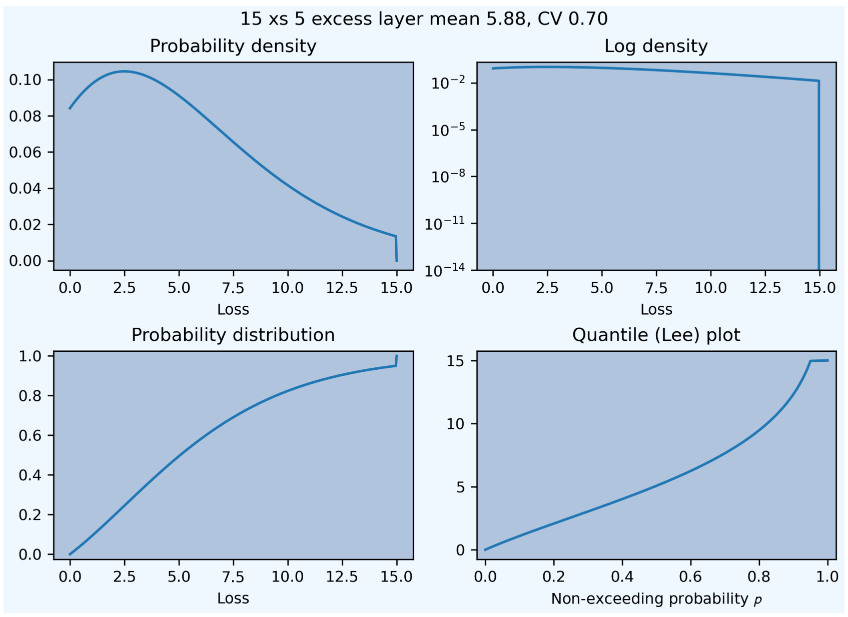

Lognorm, mean 10.00, CV 0.50, skew 1.62, kurt 5.04
Probability density
Log density
Loss
Probability distribution
Quantile (Lee) plot
Non-exceeding probability p



Examples.

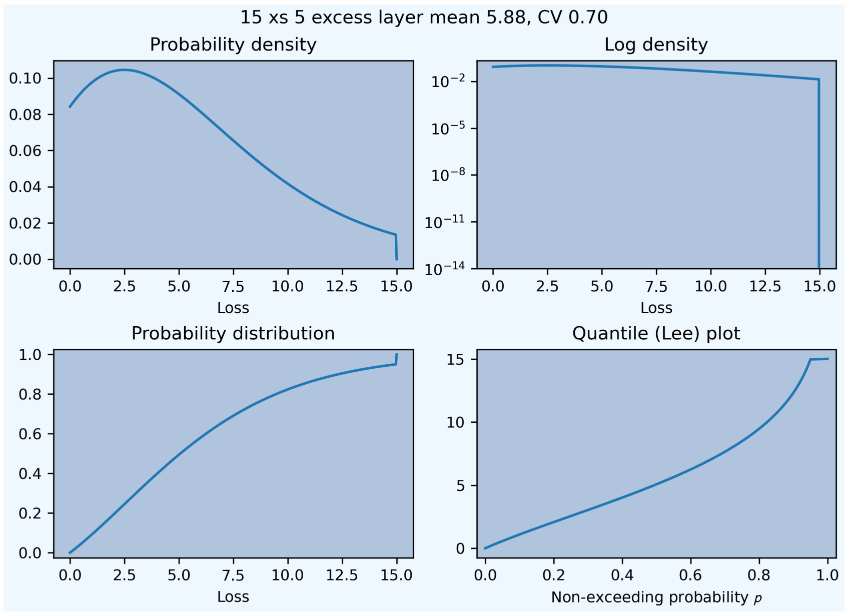
This example show the impact of adding a limit and attachment. Limits and attachments determine exposure in DecL and they belong to the `Aggregate` specification. DecL cannot be used to set the limit and attachment of a `Severity` object. One way to apply them is to create an aggregate with a fixed frequency of one claim. By default, the severity is conditional on a loss to the layer.

```
In [26]: limit, attach = 15, 5

In [27]: s2 = build(f'agg TenM:SevLayer 1 claim {limit} xs {attach} sev gamma 10␣
↪cv .5 fixed')

In [28]: m, v, sk, k = s2.sevs[0].fz.stats('mvsk')

In [29]: s2.sevs[0].plot(n=401, figsize=(2*3.5, 2*2.45+0.3), layout='AB\nCD')

In [30]: plt.gcf().suptitle(f'Ground-up severity\nGround-up gamma mean {m:.2f}, CV
↪{v**0.5/m:.2f}, skew {sk:.2f}, kurt {k:.2f}\n'
   ....:                        f'{limit} xs {attach} excess layer mean {s2.est_m:.2f},
↪ CV {s2.est_cv:.2f}, skew {s2.est_skew:.2f}, kurt {k:.2f}');
   ....:
```



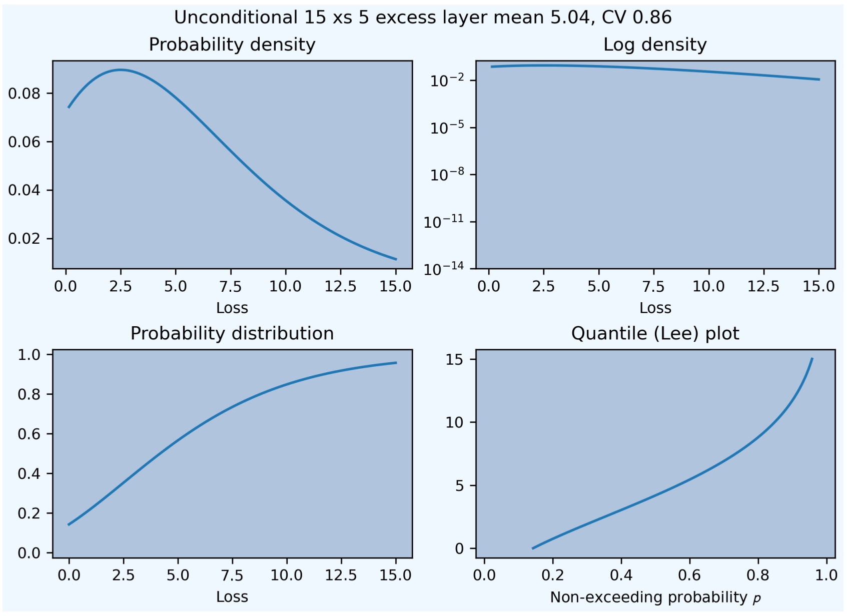
A `Severity` can be created directly using `args` and `kwargs`. Here is an example. It also shows the impact of making the severity unconditional (on a loss to the layer). Start by creating the conditional (default) severity and plotting it.

```
In [31]: from aggregate import Severity

In [32]: s3 = Severity('gamma', attach, limit, 10, 0.5)

In [33]: s3.plot(n=401, figsize=(2*3.5, 2*2.45+0.15), layout='AB\nCD')

In [34]: m, v = s3.stats()

In [35]: plt.gcf().suptitle(f'{limit} xs {attach} excess layer mean {m:.2f}, CV
↪{v**.5/m:.2f}');
```



Next, create an unconditional version. The lower pdf is scaled down by the probability of attaching the layer, and the left end of the cdf shifted up by the probability of not attaching the layer. These probabilities are given by the underlying `fz` object's sf and cdf.

```
In [36]: s4 = Severity('gamma', attach, limit, 10, 0.5, sev_conditional=False)

In [37]: s4.plot(figsize=(2*3.5, 2*2.45+0.15), layout='AB\nCD')

In [38]: m, v = s4.stats()

In [39]: plt.gcf().suptitle(f'Unconditional {limit} xs {attach} excess layer mean
↪{m:.2f}, CV {v**.5/m:.2f}');

In [40]: print(f'Probability of attaching layer {s4.fz.cdf(attach):.3f}')
Probability of attaching layer 0.143
```



Although `Severity` accepts a weight argument, it does not actually support weighted severities. It models only one component. `Aggregate` handles weighted severities by creating a separate `Severity` for each component.

2.3.5 The `Aggregate` Class

Creating an Aggregate Distribution

`Aggregate` objects can be created in three ways:

1. Generally, they are created using DecL by `Underwriter.build()`, as shown in *Object Creation Using DecL and build()*.
2. Objects in the knowledge can be created by name.
3. Advanced users and programmers can create `Aggregate` objects directly using `kwargs`, see *Aggregate Class*.

Example.

This example uses `build()` to make an `Aggregate` with a Poisson frequency, mean 5, and gamma severity with mean 10 and CV 1 . It includes more discussion than the example above. The line breaks improve readability but are cosmetic.

```
In [41]: a02 = build('agg TenM:02 '
   ....:             '5 claims '
   ....:             'sev gamma 10 cv 1 '
```

(continues on next page)

(continued from previous page)

```
   ....:                'poisson')
   ....:

In [42]: qd(a02)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     5                       0.44721            0.44721
Sev     10       10 -2.5431e-08       1         1        2           2
Agg     50       50 -2.5432e-08 0.63246   0.63246  0.94868     0.94868
log2 = 16, bandwidth = 1/128, validation: not unreasonable.
```

qd displays the dataframe a.describe. This example fails the aliasing validation test because the aggregate mean error is suspiciously greater than the severity. (Run with logger level 20 for more diagnostics.) However, it passes both the severity mean and aggregate mean tests.

Aggregate Quick Diagnostics

The quick display reports a set of quick diagnostics, showing

- Exact E[X] and estimated Est E[X] frequency, severity, and aggregate statistics.
- Relative errors Err E[X] for the means.
- Coefficient of variation CV(X) and estimated CV, Est CV(X)
- Skewness Skew(X) and estimated skewness, Est Skew(X)

The line below the table shows the (log to base 2) of the number of buckets used, log2 and the bucket size bs used in discretization.

These statistics make it easy to see if the numerical estimation is invalid. Look for a small error in the mean and close second (CV) and third (skew) moments. The last item validation: not unreasonable shows the model did not fail any tests. The test should be interpreted like a null hypothesis; you expect it to be True and are worried when it is False.

In this case, the aggregate mean error is too high because the discretization bucket size bs is too small. Update with a larger bucket.

```
In [43]: a02.update(bs=1/128)

In [44]: qd(a02)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     5                       0.44721            0.44721
Sev     10       10 -2.5431e-08       1         1        2           2
Agg     50       50 -2.5432e-08 0.63246   0.63246  0.94868     0.94868
log2 = 16, bandwidth = 1/128, validation: not unreasonable.
```

Aggregate Algorithm in Detail

Here's the `aggregate` FFT convolution algorithm stripped down to bare essentials and coded in raw Python to show you what happens behind the curtain. The algorithm steps are:

1. Inputs
   - Severity distribution cdf. Use `scipy.stats`.
   - Frequency distribution probability generating function. For a Poisson with mean λ the PGF is $\mathscr{P}(z) = \exp(\lambda(z-1))$.
   - The bucket size b. Use the value `simple.bs`.
   - The number of buckets $n = 2^{log_2}$. Use the default `log2=16` found in `simple.log2`.
   - A padding parameter, equal to 1 by default, from `simple.padding`.
2. Discretize the severity cdf.
3. Apply the FFT to discrete severity with padding to size `2**(log2 + padding)`.
4. Apply the frequency pgf to the FFT.
5. Apply the inverse FFT to create is a discretized version of the aggregate distribution and output it.

Let's recreate the following simple example. The variable names for the means and shape are for clarity. `sev_shape` is σ for a lognormal.

```
In [45]: from aggregate import build, qd

In [46]: en = 50

In [47]: sev_scale = 10

In [48]: sev_shape = 0.8

In [49]: simple = build('agg Simple '
   ....:                f'{en} claims '
   ....:                f'sev {sev_scale} * lognorm {sev_shape} '
   ....:                'poisson')
   ....:

In [50]: qd(simple)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     50                     0.14142            0.14142
Sev  13.771   13.771 -2.4063e-09 0.94683   0.94683   3.6893      3.6892
Agg  688.56   688.56 -9.4705e-09 0.19476   0.19476  0.36935     0.36933
log2 = 16, bandwidth = 1/32, validation: not unreasonable.
```

The next few lines of code implement the FFT convolution algorithm. Start by importing the probability distribution and FFT routines. `rfft` and `irfft` take the FFT and inverse FFT of real input.

```
In [51]: import numpy as np

In [52]: from scipy.fft import rfft, irfft

In [53]: import scipy.stats as ss
```

Pull parameters from `simple` to match calculations, step 1. `n_pad` is the length of the padded vector used in the convolution to manage aliasing.

```
In [54]: bs = simple.bs

In [55]: log2 = simple.log2

In [56]: padding = simple.padding

In [57]: n = 1 << log2

In [58]: n_pad = 1 << (log2 + padding)

In [59]: sev = ss.lognorm(sev_shape, scale=sev_scale)
```

Use the `round` method and the survival function to discretize, completing step 2.

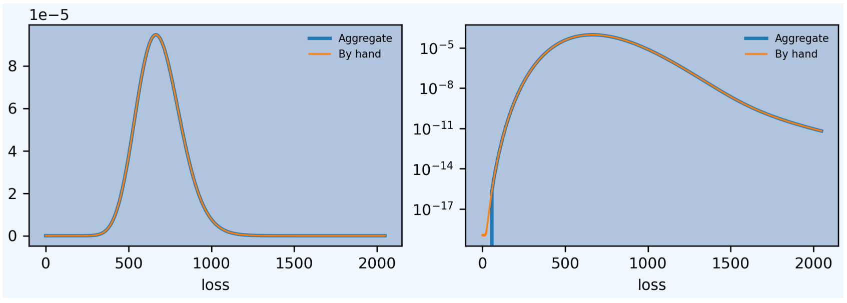
```
In [60]: xs = np.arange(0, (n + 1) * bs, bs)

In [61]: discrete_sev = -np.diff(sev.sf(xs - bs / 2))
```

The next line of code carries out algorithm steps 3, 4, and 5! All the magic happens here. The forward FFT adds padding, but the answer must be unpadded manually, with the final `[:n]`.

```
In [62]: agg = irfft( np.exp( en * (rfft(discrete_sev, n_pad) - 1) ) )[:n]
```

Plots to compare the two approaches. They are spot on!

```
In [63]: import matplotlib.pyplot as plt

In [64]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45),
   ....:     constrained_layout=True);                                    \
   ....: ax0, ax1 = axs.flat;                                             \
   ....: simple.density_df.p_total.plot(lw=2, label='Aggregate', ax=ax0); \
   ....: ax0.plot(xs[:-1], agg, lw=1, label='By hand');                   \
   ....: ax0.legend();                                                    \
   ....: simple.density_df.p_total.plot(lw=2, label='Aggregate', ax=ax1); \
   ....: ax1.plot(xs[:-1], agg, lw=1, label='By hand');                   \
   ....: ax1.legend();
   ....:

In [65]: ax1.set(yscale='log');
```



The very slight difference for small loss values arises because `build` removes numerical fuzz, setting values below machine epsilon (about `2e-16`) to zero, explaining why the blue aggregate line drops off vertically on the left.

Basic Probability Functions

An `Aggregate` object acts like a discrete probability distribution. There are properties for the aggregate and severity mean, standard deviation, coefficient of variation, and skewness, both computed exactly and numerically estimated.

```
In [66]: print(a02.agg_m, a02.agg_sd, a02.agg_cv, a02.agg_skew)
50.0 31.622776601683793 0.6324555320336759 0.9486832980505139

In [67]: print(a02.est_m, a02.est_sd, a02.est_cv, a02.est_skew)
49.9999987284203 31.622777003581923 0.6324555561559914 0.9486832857144013

In [68]: print(a02.sev_m, a02.sev_sd, a02.sev_cv, a02.sev_skew)
10.0 10.0 1.0 2.0

In [69]: print(a02.est_sev_m, a02.est_sev_sd, a02.est_sev_cv, a02.est_sev_skew)
9.99999974568674 10.000000508623474 1.0000000762936754 1.9999996947979108
```

They have probability mass, cumulative distribution, survival, and quantile (inverse of distribution) functions.

```
In [70]: a02.pmf(60), a02.cdf(50), a02.sf(60), a02.q(a02.cdf(60)), a02.q(0.5)
Out[70]: (7.923645058165983e-05, 0.5639640504996987, 0.3244107518264777, 60.0, 44.
↪90625)
```

The pdf, cdf, and sf for the underlying severity are also available.

```
In [71]: a02.sev.pdf(60), a02.sev.cdf(50), a02.sev.sf(60)
Out[71]: (0.00024787521766663585, 0.9932620530009145, 0.002478752176666357)
```

Note: `Aggregate` and `Portfolio` objects need to be updated after they have been created. Updating builds out discrete numerical approximations, analogous to simulation. By default, `build()` handles updating automatically.

Warning: Always use bucket sizes that have an exact binary representation (integers, 1/2, 1/4, 1/8, etc.) **Never** use 0.1 or 0.2 or other numbers that do not have an exact float representation, see REF.

Mixtures

An `Aggregate` can have a mixed severity. The mixture can include different distributions, parameters, shifts, and locations.

```
In [72]: a03 = build('agg TenM:03 '
   ....:             '25 claims '
   ....:             'sev [gamma lognorm invgamma] [5 10 10] cv [0.5 0.75 1.5] '
   ....:             '+ [0 10 20] wts [.5 .25 .25] '
   ....:             'mixed gamma 0.5'
   ....:            , bs=1/16)
   ....:

In [73]: qd(a03)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    25                      0.53852             1.0028
Sev     15   14.999 -3.7467e-05 0.90907   0.89065      inf      9.9075
Agg    375   374.99 -3.8034e-05 0.56838    0.5672      inf      1.0698
log2 = 16, bandwidth = 1/16, validation: fails sev cv, agg cv.
```

An `Aggregate` can model multiple units at once, and allow them to share mixing variables. This induces correlation between the components, see the *report dataframe*. All parts of the specification can vary, including limits and attachments (not shown). This case differentiated from a mixed severity by having no weights.

```
In [74]: a04 = build('agg TenM:04 '
   ....:             '[500 250 100] premium at [.8 .7 .5] lr '
   ....:             'sev [gamma lognorm invgamma] [5 10 10] cv [0.5 0.75 1.5] '
   ....:             'mixed gamma 0.5'
   ....:            , bs=1/8)
   ....:

In [75]: qd(a04)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  102.5                      0.50966             1.0002
Sev   6.0976    6.0975 -6.5732e-06 0.89437   0.87819      inf      43.583
Agg      625       625 -6.6355e-06 0.51726   0.51699      inf      1.0208
log2 = 16, bandwidth = 1/8, validation: fails sev cv.
```

Accessing Severity in an `Aggregate`

The attribute `Aggregate.sevs` is an array of the `Severity` objects. Usually, it contains only one distribution but when severity is a mixture it contains one for each mixture component. It can be iterated over. Each `Severity` object wraps a `scipy.stats` continuous random variable called `fz` that represents ground-up severity. The `args` are its shape variable(s) and `kwds` its scale and location variables. This is most interesting when the object has a mixed severity.

```
In [76]: for s in a03.sevs:
   ....:     print(s.sev_name, s.fz.args, s.fz.kwds)
   ....:
gamma (4.0,) {'scale': 1.25, 'loc': 0.0}
lognorm (0.6680472308365775,) {'scale': 8.0, 'loc': 10.0}
invgamma (2.4444444444444446,) {'scale': 14.444444444444446, 'loc': 20.0}
```

The property `a03.sev` is a `namedtuple` exposing the exact weighted pdf, cdf, and sf of the underlying `Severity` objects.

```
In [77]: a03.sev.pdf(20), a03.sev.cdf(20), a03.sev.sf(20)
Out[77]: (0.014150102336136361, 0.657658211871339, 0.342341788128661)
```

The component weights are proportional to `a03.en` and `a03.sev.cdf` is computed as

```
In [78]: (np.array([s.cdf(20) for s in a03.sevs]) * a03.en).sum() / a03.en.sum()
Out[78]: 0.6576582118713391
```

The following are equal using the defaut discretization method.

```
In [79]: a03.density_df.loc[20, 'F_sev'], a03.sev.cdf(20 + a03.bs/2)
Out[79]: (0.6580993598330426, 0.6580993598330431)
```

Reinsurance

`Aggregate` objects can apply per occurrence and aggregate reinsurance using clauses

- `occurrence net of [limit] xs ]attach]`
- `occurrence net of [pct] so [limit] xs [attach]`, where `so` stands for "share of"
- `occurrence ceded to [limit] xs ]attach]`
- and so forth.

Examples.

Gross distribution: a triangular aggregate created as the sum of two uniform distribution on 1, 2,..., 10.

```
In [80]: a05g = build('agg TenM:05g dfreq [2] dsev [1:10]')

In [81]: qd(a05g)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     2                            0
Sev    5.5      5.5           0 0.52223   0.52223       0           0
Agg     11       11 -3.3307e-16 0.36927   0.36927       0 -6.1064e-14
log2 = 6, bandwidth = 1, validation: not unreasonable.
```

Apply 3 xs 7 occurrence reinsurance to cap individual losses at 7. `a05no` is the net of occurrence distribution.

```
In [82]: a05no = build('agg TenM:05no dfreq [2] dsev [1:10] '
   ....:               'occurrence net of 3 xs 7')
   ....:

In [83]: qd(a05no)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     2                         0
Sev    5.5      4.9 -0.10909 0.52223   0.44197       0    -0.52103
Agg     11      9.8 -0.10909 0.36927   0.31252       0    -0.36842
log2 = 6, bandwidth = 1, validation: n/a, reinsurance.
```

Warning: The `describe` dataframe always reports gross analytic statistics (`E[X]`, `CV(X)`, `Skew(X)`) and the requested net or ceded estimated statistics (`Est E[X]`, `Est CV(X)`, `Est Skew(X)`). Look at the gross portfolio first to check computational accuracy. Net and ceded "error" report the difference to analytic gross.

Add an aggregate 4 xs 8 reinsurance cover on the net of occurrence distribution. `a05n` is the final net distribution.

```
In [84]: a05n = build('agg TenM:05n dfreq [2] dsev [1:10] '
   ....:              'occurrence net of 3 xs 7 '
   ....:              'aggregate net of 4 xs 8')
   ....:

In [85]: qd(a05n)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     2                         0
Sev    5.5      4.9 -0.10909 0.52223   0.44197       0    -0.52103
Agg     11     7.84 -0.28727 0.36927   0.20781       0     -1.2676
log2 = 6, bandwidth = 1, validation: n/a, reinsurance.
```

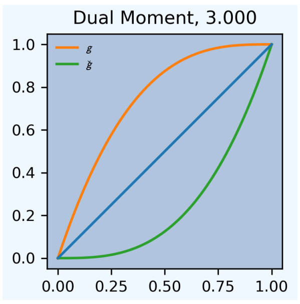
See *The plot() Method* for plots of the different distributions.

2.3.6 The `Distortion` Class

See *Distortions and Spectral Risk Measures* and PIR Chapter 10.5 for more information about distortions.

A `Distortion` can be created using DecL. It object has methods for `g`, the distortion function, along with its dual `g_dual(s)=1-g(1-s)` and inverse `g_inv`. The `plot()` method shows `g` (above the diagonal) and `g_inv` (below).

```
In [86]: d06 = build('distortion TenM:06 dual 3')

In [87]: qd(d06.g(.2), d06.g_inv(.2), d06.g_dual(0.2),
   ....: d06.g(.8), d06.g_inv(.992), d06)
   ....:
0.488
0.071682
0.008
0.992
0.8
Dual Moment, 3.000

In [88]: d06.plot();
```



The `Distortion` class can create distortions from a number of parametric families.

```
In [89]: from aggregate import Distortion

In [90]: Distortion.available_distortions(False, False)
Out[90]:
('ph',
 'wang',
 'cll',
 'lep',
 'ly',
 'clin',
 'dual',
 'ccoc',
 'tvar',
 'bitvar',
 'convex',
 'tt')
```

Run the command:

```
Distortion.test()
```

for graphs of samples from each available family. `tt` is not a distortion because it is not concave. It is included for historical reasons.

2.3.7 The `Portfolio` Class

A `Portfolio` object models a portfolio (book, block) of units (accounts, lines, business units, regions, profit centers), each represented as an `Aggregate`. It uses FFTs to convolve (add) the unit distributions. By default, all the units are assumed to be independent, though there are ways to adjust this. REF. The independence assumption is not as bad as it may appear; its effect can be ameliorated by selecting units carefully and sharing mixing variables appropriately (see REF for further discussion).

`Portfolio` objects have all of the attributes and methods of a `Aggregate` and add methods for pricing and allocation to units.

The DecL for a portfolio is simply:

```
port NAME AGG1 <AGG2> <AGG3> ...
```

where `AGG1` is an aggregate specification. Portfolios can have one or more units. The DecL can be split over multiple lines if each aggregate begins on a new line and is indented by a tab (like a Python function).

Example.

Here is a three-unit portfolio built using a DecL program. The line breaks and horizontal spacing are cosmetic since Python just concatenates the input.

```
In [91]: p07 = build('port TenM:07 '
   ....:             'agg A '
   ....:                 '100 claims '
   ....:                 '10000 xs 0 '
   ....:                 'sev lognorm 100 cv 1.25 '
   ....:                 'poisson '
   ....:             'agg B '
   ....:                 '150 claims '
   ....:                 '2500 xs 5 '
   ....:                 'sev lognorm 50 cv 0.9 '
   ....:                 'mixed gamma .6 '
   ....:             'agg Cat '
   ....:                 '2 claims '
   ....:                 '1e5 xs 0 '
   ....:                 'sev 500 * pareto 1.8 - 500 '
   ....:                 'poisson'
   ....:            , approximation='exact', padding=2)
   ....:

In [92]: qd(p07)

             E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
unit  X
A     Freq    100                           0.1                 0.1
      Sev     100       100 -1.1973e-08  1.2498     1.2498   5.6619       5.6617
      Agg   10000     10000 -1.1973e-08 0.16006    0.16007   0.4082      0.40819
B     Freq    150                       0.60553              1.2001
      Sev  45.213    45.212 -1.2825e-05 0.99517    0.99528   3.4345       3.4335
      Agg  6781.9    6781.8 -1.2825e-05 0.61096    0.61096   1.2009       1.2009
Cat   Freq      2                       0.70711             0.70711
      Sev  616.02    616.02 -9.7399e-07  3.1331     3.1331   23.278       23.278
      Agg    1232      1232 -4.3508e-06  2.3256     2.3254   14.837       14.828
total Freq    252                       0.36266              1.1783
      Sev  71.484    71.483 -4.9016e-06  2.7998              172.51
      Agg   18014     18013 -2.6963e-05 0.29343    0.29322   2.9528        2.925
log2 = 16, bandwidth = 2, validation: not unreasonable.
```

The portfolio units are called A, B and Cat. Printing using `qd` shows `p07.describe`, which concatenates each unit's `describe` and adds the same statistics for the total.

- Unit A has 100 (expected) claims, each pulled from a lognormal distribution with mean of 30 and coefficient of variation 1.25 within the layer 100 xs 0 (i.e., losses are limited at 100). The frequency distribution is Poisson.
- Unit B is similar.
- The Cat unit is has expected frequency of 2 claims from the indicated limit, with severity given by a Pareto distribution with shape parameter 1.8, scale 500, shifted left by 500. This corresponds to the usual Pareto with survival function $S(x) = (500/(500 + x))^{1.8} = (1 + x/500)^{-1.8}$ for $x \geq 0$.

The portfolio total (i.e., the sum of the units) is computed using FFTs to convolve (add) the unit's aggregate distributions. All computations use the same discretization bucket size; here the bucket-size `bs=2`. See For Portfolio Objects.

A `Portfolio` object acts like a discrete probability distribution, the same as an `Aggregate`. There are properties for the mean, standard deviation, coefficient of variation, and skewness, both computed exactly and numerically estimated.

```
In [93]: print(p07.agg_m, p07.agg_sd, p07.agg_cv, p07.agg_skew)
18013.930242377213 5285.799078302742 0.29342841940555886 2.9527686953572805

In [94]: print(p07.est_m, p07.est_sd, p07.est_cv, p07.est_skew)
18013.444535983497 5281.917148512574 0.29322082947331163 2.9249704103192653
```

They have probability mass, cumulative distribution, survival, and quantile (inverse of distribution) functions.

```
In [95]: p07.pmf(12000), p07.cdf(11000), p07.sf(12000), p07.q(p07.cdf(12000)), p07.
↪q(0.5)
Out[95]:
(9.80844644134903e-05,
 0.031240713459624474,
 0.930037600482374,
 12000.0,
 17176.0)
```

The names of the units in a `Portfolio` are in a list called `p07.unit_names` or `p07.unit_names_ex` including `total`. The `Aggregate` objects in the `Portfolio` can be iterated over.

```
In [96]: for u in p07:
   ....:     print(u.name, u.agg_m, u.est_m)
   ....:
A 9999.982520635798 9999.982400907693
B 6781.909658669757 6781.822680927324
Cat 1232.038063071656 1232.0327026845005
```

2.3.8 Estimating Bucket Size for Discretization

Selecting an appropriate bucket size `bs` is critical to obtaining accurate results. This is a hard problem that may have hindered broad adoption of FFT-based methods.

See *Numerical Methods and FFT Convolution* for further discussion.

Hyper-parameters `log2` and `bs`

The hyper-parameters `log2` and `bs` control numerical calculations. `log2` equals the log to base 2 of the number of buckets used and `bs` equals the bucket size. These values are printed by `qd`.

Estimating and Testing `bs` For `Aggregate` Objects

For an `Aggregate`, `recommend_bucket()` uses a shifted lognormal method of moments fit and takes the `recommend_p` percentile as the right-hand end of the discretization. By default `recommend_p=0.999`, but for thick tailed distributions it may be necessary to use a value closer to 1. `recommend_bucket()` also considers any limits: ideally limits are multiples of the bucket size.

The recommended value of `bs` should rounded up to a binary fraction (denominator is a power of 2) using `utilities.round_bucket()`.

`Aggregate` also includes two functions for assessing `bs`, one based on the overall error and one based on looking at each severity component.

`Aggregate.aggregate_error_analysis()` updates the object at a range of different `bs` values and reports the total absolute (strictly, signed absolute error) and relative error as well as an upper bound `bs/2` on the absolute value of the discretization error. `log2` must be input and, optionally, the log base 2 of the smallest bucket to model. It then models six doublings of the input bucket. If no bucket is input, it models three doublings up and down from the rounded `recommend_bucket()` suggestion. The output table shows:

- The actual `(agg, m)` and estimated `(est, m)` means, from the `describe` dataframe.
- The implied absolute `(abs, m)` and relative `(rel, m)` errors in the mean.
- `(rel, h)` shows the maximum relative severity discretization error, which equals `bs / 2` divided by the average severity.
- `(rel, total)`, equal to the sum of `(rel, h)` and `rel m`.

Thick tailed distributions can favor a large bucket size without regard to the impact on discretization; accounting for the impact of `bs / 2` is a countervailing force.

```
In [97]: qd(a04.aggregate_error_analysis(16), sparsify=False, col_space=9)

view              agg       est         abs         rel       rel        rel
stat                m         m           m           m         h      total
bs
0.03125           625    624.71    -0.28786 -0.00046057 0.0025625 -0.0030231
0.06250           625    624.99   -0.011699 -1.8718e-05  0.005125 -0.0051437
0.12500           625       625  -0.0041472 -6.6355e-06   0.01025  -0.010257
0.25000           625       625  -0.0015133 -2.4212e-06    0.0205  -0.020502
0.50000           625       625 -0.00050174 -8.0278e-07     0.041  -0.041001
1.00000           625       625   0.0026758  4.2813e-06     0.082   0.082004
2.00000           625    625.14     0.13828  0.00022125     0.164    0.16422
```

`Aggregate.severity_error_analysis()` performs a detailed error analysis of each severity component. It reports:

- The name, limit, attachment, and truncation point for each severity component.
- `S` the probability the component (or total losses) exceed the truncation.
- `sum_p` the sum of discrete probabilities, which can be < 1 if `normalize=False`.

- `wt` the weight of the component and `en` the corresponding claim count.
- `agg_mean` and `agg_wt` the aggregate mean contribution from the component (sums to the overall mean), and the each component's proportion of the total. The loss weight can differ drastically from the count weight.
- `mean` and `est_mean` the analytic and estimated severity by component and the corresponding `abs` and `rel` error.
- `trunc_error` the truncation error by component (tail integral) and relative truncation error.
- The `h_error` based on `bs / 2` by component, a (conservative) upper bound on discretization error and the relative error compared to the component mean.
- `h2_adj` and `rel_h2_adj` estimate a second order adjustment to the numerical mean. They give a better idea of the discretization error.

```
In [98]: qd(a04.severity_error_analysis(), line_width=75)

       name  limit  attachment  trunc           S  sum_p       wt    en  \
0     gamma    inf           0   8192           0      1 0.78049    80
1   lognorm    inf           0   8192 1.5989e-25      1 0.17073  17.5
2  invgamma    inf           0   8192 5.9297e-08      1 0.04878     5
3     total    inf           0   8192 8.6388e-08      1        1 102.5

   agg_mean  agg_wt   mean  est_mean         abs         rel  \
0       400    0.64      5         5  1.5997e-10  3.1994e-11
1       175    0.28     10        10 -1.6414e-12 -1.6414e-13
2        50    0.08     10    9.9992 -0.00082165 -8.2165e-05
3       625       1 6.0976    6.0975   -4.008e-05 -6.5732e-06

   trunc_error  rel_trunc_error  h_error  rel_h_error     h2_adj  \
0           -0               -0   0.0625       0.0125 5.1607e-09
1  -8.8373e-23      -8.8373e-24   0.0625      0.00625 1.0915e-14
2  -0.00033646      -3.3646e-05   0.0625      0.00625 1.1514e-14
3  -1.6408e-05      -2.6909e-06   0.0625      0.01025 4.0279e-09

   rel_h2_adj
0  1.0321e-09
1  1.0915e-15
2  1.1514e-15
3  6.6058e-10
```

Generally there is either discretization or truncation error. Look for one of them to dominate. Discretization error is solved with a smaller bucket; truncation with a larger. When the two conflict, add more buckets by increasing `log2`.

Estimating and Testing `bs` For `Portfolio` Objects

For a `Portfolio`, the right hand end of the distribution is estimated using the square root of sum of squares (proxy independent sum) of the right hand ends of each unit.

The method `port.recommend_bucket()` suggests a reasonable bucket size.

```
In [99]: print(p07.recommend_bucket().iloc[:, [0,3,6,10]])
             bs10       bs13      bs16      bs20
line
A       18.545709   2.318214  0.289777  0.018111
B       43.766975   5.470872  0.683859  0.042741
Cat    143.204295  17.900537  2.237567  0.139848
total  205.516978  25.689622  3.211203    0.2007

In [100]: p07.best_bucket(16)
Out[100]: 2
```

The column bsN corresponds to discretizing with 2**N buckets. The rows show suggested bucket sizes by unit and in total. For example with N=16 (i.e., 65,536 buckets) the suggestion is 2.19. It is best the bucket size is a divisor of any limits or attachment points. best_bucket() takes this into account and suggests 2.

To test bs, run the tests above on each unit.

2.3.9 Methods and Properties Common To Aggregate and Portfolio Classes

Aggregate and Portfolio both have the following methods and properties. See *Aggregate Class* and *Portfolio Class* for full lists.

- info and describe are dataframes with statistics and other information; they are printed with the object.
- density_df a dataframe containing estimated probability distributions and other expected value information.
- The statistics dataframe shows analytically computed mean, variance, CV, and skewness for each unit and in total.
- report_df are dataframe with information to test if the numerical approximations appear valid. Numerically estimated statistics are prefaced est_ or empirical.
- log2 and bs hyper-parameters that control numerical calculations.
- spec a dictionary containing the kwargs needed to recreate each object. For example, if a is an Aggregate object, then Aggregate(**a.spec) creates a new copy.
- spec_ex a dictionary that appends hyper-parameters to spec including log2 and bs.
- program the DecL program used to create the object. Blank if the object has been created directly. (A given object can often be created in different ways by DecL, so there is no obvious reverse mapping from the spec.)
- renamer a dictionary used to rename columns of member dataframes to be more human readable.
- update() a method to run the numerical calculation of probability distributions.
- recommend_bucket() to recommend the value of bs.
- Common statistical functions including pmf, cdf, sf, the quantile function (value at risk) and tail value at risk.
- Statistical functions: pdf, cdf, sf, quantile, value at risk, tail value at risk, and so on.
- plot() method to visualize the underlying distributions. Plots the pmf and log pmf functions and the quantile function. All the data is contained in density_df and the plots are created using pandas standard plotting commands.
- price() to apply a distortion (spectral) risk measure pricing rule with a variety of capital standards.
- snap() to round an input number to the index of density_df.
- approximate() to create an analytic approximation.
- sample() pulls samples, see *Samples from aggregate Object*.

The info Dataframe

The info dataframe contains information about the frequency and severity stochastic models, how the object was computed, and any reinsurance applied (none in this case).

```
In [101]: print(a05n.info)
aggregate object name    TenM:05n
claim count              2.00
frequency distribution   empirical
severity distribution    dhistogram, unlimited.
bs                       1
log2                     6
```

(continues on next page)

(continued from previous page)

```
padding                      1
sev_calc                     discrete
normalize                    True
approximation                exact
validation_eps               0.0001
reinsurance                  occurrence and aggregate
occurrence reinsurance       net of 100% share of 3 xs 7 per occurrence
aggregate reinsurance        net of 100% share of 4 xs 8 in the aggregate.
validation                   n/a, reinsurance


In [102]: print(p07.info)
portfolio object name        TenM:07
aggregate objects            3
bs                           2
log2                         16
padding                      2
sev_calc                     discrete
normalize                    True
last update                  2024-03-06T22:23:30
hash                         73bb77ccfd1b4cd1
```

The `describe` Dataframe

The `describe` dataframe contains gross analytic and estimated (net or ceded) statistics. When there is no reinsurance, comparison of analytic and estimated moments provides a test of computational accuracy (first case). It should always be reviewed after updating. When there is reinsurance, empirical is net (second case).

```
In [103]: qd(a05g.describe)

      E[X]  Est E[X]     Err E[X]   CV(X)  Est CV(X)  Err CV(X)  Skew(X)  Est␣
↪Skew(X)
X                                                                              ␣
↪
Freq     2       NaN          NaN       0        NaN        NaN      NaN      ␣
↪NaN
Sev    5.5       5.5            0 0.52223    0.52223          0        0       ␣
↪0
Agg     11        11 -3.3307e-16 0.36927    0.36927          0        0  -6.1064e-
↪14

In [104]: with pd.option_context('display.max_columns', 15):
   .....:     print(a05n.describe)
   .....:
      E[X]  Est E[X]  Err E[X]     CV(X)  Est CV(X)  Err CV(X)  Skew(X)  Est␣
↪Skew(X)
X                                                                              ␣
↪
Freq   2.0       NaN       NaN  0.000000        NaN        NaN      NaN      ␣
↪NaN
Sev    5.5      4.90 -0.109091  0.522233   0.441968  -0.153697      0.0    -0.
↪521027
Agg   11.0      7.84 -0.287273  0.369274   0.207810  -0.437247      0.0    -1.
↪267573

In [105]: qd(p07.describe)

             E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)   Err CV(X)  Skew(X) ␣
↪Est Skew(X)
```

(continues on next page)

(continued from previous page)

```
unit  X                                                                          ␣
↪
A     Freq    100         NaN          NaN      0.1         NaN          NaN       0.1   ␣
↪      NaN
      Sev     100         100 -1.1973e-08   1.2498      1.2498  1.0689e-05    5.6619   ␣
↪    5.6617
      Agg   10000       10000 -1.1973e-08  0.16006     0.16007  6.5169e-06    0.4082   ␣
↪   0.40819
B     Freq    150         NaN          NaN  0.60553         NaN          NaN    1.2001   ␣
↪      NaN
      Sev   45.213     45.212 -1.2825e-05  0.99517     0.99528  0.00010815    3.4345   ␣
↪    3.4335
      Agg   6781.9     6781.8 -1.2825e-05  0.61096     0.61096   1.913e-06    1.2009   ␣
↪    1.2009
Cat   Freq      2         NaN          NaN  0.70711         NaN          NaN   0.70711   ␣
↪      NaN
      Sev   616.02     616.02 -9.7399e-07   3.1331      3.1331   1.118e-06    23.278   ␣
↪    23.278
      Agg     1232       1232 -4.3508e-06   2.3256      2.3254  -7.109e-05    14.837   ␣
↪    14.828
total Freq    252         NaN          NaN  0.36266         NaN          NaN    1.1783   ␣
↪      NaN
      Sev   71.484     71.483 -4.9016e-06   2.7998         NaN          NaN    172.51   ␣
↪      NaN
      Agg    18014      18013 -2.6963e-05  0.29343     0.29322 -0.00070746    2.9528   ␣
↪     2.925
```

Printing the object using `qd` add `log2`, `bs`, and validation information.

The `density_df` Dataframe

The `density_df` dataframe contains a wealth of information. It has `2**log2` rows and is indexed by the outcomes, all multiples of `bs`. Columns containing `p` are the probability mass function, of the aggregate or severity.

- the aggregate and severity pmf (called `p` and duplicated as `p_total` for consistency with `Portfolio` objects), log pmf, cdf and sf
- the aggregate lev (duplicated as `exa`)
- `exlea` (less than or equal to `a`) which equals $\mathsf{E}[X \mid X \le a]$ as a function of `loss`
- `exgta` (greater than) which equals $\mathsf{E}[X \mid X > a]$

In an `Aggregate`, `p` and `p_total` are identical, the latter included for consistency with `Portfolio` output. `F` and `S` are the cdf and sf (survival function). `lev` and `exa` are identical and equal the limited expected value at the `loss` level. Here are the first five rows.

```
In [106]: print(a05g.density_df.shape)
(64, 17)

In [107]: print(a05g.density_df.columns)
Index(['loss', 'p_total', 'p', 'p_sev', 'log_p', 'log_p_sev', 'F', 'F_sev', 'S',
↪'S_sev', 'lev', 'exa', 'exlea', 'e',
       'epd', 'exgta', 'exeqa'],
      dtype='object')

In [108]: with pd.option_context('display.max_columns', a05g.density_df.shape[1]):
   .....:     print(a05g.density_df.head())
   .....:
      loss  p_total      p  p_sev      log_p  log_p_sev     F  F_sev      S  S_sev  ␣
↪lev   exa      exlea      e        epd  \
```

(continues on next page)

(continued from previous page)

```
loss                                                                                       ␣
↪
0.0     0.0      0.00  0.00    0.0      -inf        -inf  0.00    0.0  1.00    1.0  0.
↪00  0.00       NaN  11.0  1.000000
1.0     1.0      0.00  0.00    0.1      -inf   -2.302585  0.00    0.1  1.00    0.9  1.
↪00  1.00       NaN  11.0  0.909091
2.0     2.0      0.01  0.01    0.1 -4.605170  -2.302585  0.01    0.2  0.99    0.8  2.
↪00  2.00  2.000000  11.0  0.818182
3.0     3.0      0.02  0.02    0.1 -3.912023  -2.302585  0.03    0.3  0.97    0.7  2.
↪99  2.99  2.666667  11.0  0.728182
4.0     4.0      0.03  0.03    0.1 -3.506558  -2.302585  0.06    0.4  0.94    0.6  3.
↪96  3.96  3.333333  11.0  0.640000

          exgta  exeqa
loss
0.0   11.000000    0.0
1.0   11.000000    1.0
2.0   11.090909    2.0
3.0   11.257732    3.0
4.0   11.489362    4.0
```

The `Portfolio` version is more exhaustive. It includes a variety of columns for each unit, suffixed `_unit`, and for the complement of each unit (sum of everything but that unit) suffixed `_ημ_unit`. The totals are suffixed `_total`. The most important columns are `exeqa_unit`, *Conditional Expected Values*. All the column names and a subset of `density_df` are shown next.

```
In [109]: print(p07.density_df.shape)
(65536, 46)

In [110]: print(p07.density_df.columns)
Index(['loss', 'p_A', 'p_B', 'p_Cat', 'p_total', 'F', 'S', 'exa_total', 'lev_total
↪', 'exlea_total', 'e_total',
       'exgta_total', 'exeqa_total', 'exeqa_A', 'lev_A', 'exlea_A', 'e_A', 'exgta_A
↪', 'exi_x_A', 'exi_xlea_A',
       'exi_xgta_A', 'exi_xeqa_A', 'exa_A', 'exeqa_B', 'lev_B', 'exlea_B', 'e_B',
↪'exgta_B', 'exi_x_B', 'exi_xlea_B',
       'exi_xgta_B', 'exi_xeqa_B', 'exa_B', 'exeqa_Cat', 'lev_Cat', 'exlea_Cat',
↪'e_Cat', 'exgta_Cat', 'exi_x_Cat',
       'exi_xlea_Cat', 'exi_xgta_Cat', 'exi_xeqa_Cat', 'exa_Cat', 'exi_xlea_sum',
↪'exi_xgta_sum', 'exi_xeqa_sum'],
      dtype='object')

In [111]: with pd.option_context('display.max_columns', p07.density_df.shape[1]):
   .....:     print(p07.density_df.filter(regex=r'[aipex012]_A').head())
   .....:
     p_A  exeqa_A  exlea_A          e_A       exgta_A   exi_x_A  exi_xlea_A  exi_
↪xgta_A  exi_xeqa_A     exa_A
0.0  0.0      0.0      0.0  9999.982401  9999.982401  0.585913         0.0    0.
↪585913        0.0  0.000000
2.0  0.0      0.0      0.0  9999.982401  9999.982401  0.585913         NaN    0.
↪585913        0.0  1.171827
4.0  0.0      0.0      0.0  9999.982401  9999.982401  0.585913         NaN    0.
↪585913        0.0  2.343653
6.0  0.0      0.0      0.0  9999.982401  9999.982401  0.585913         NaN    0.
↪585913        0.0  3.515480
8.0  0.0      0.0      0.0  9999.982401  9999.982401  0.585913         NaN    0.
↪585913        0.0  4.687306
```

The `statistics` Series and Dataframe

The `statistics` dataframe shows analytically computed mean, variance, CV, and skewness. It is indexed by

- severity name, limit and attachment,
- `freq1, freq2, freq3` non-central frequency moments,
- `sev1, sev2, sev3` non-central severity moments, and
- the mean, cv and skew(ness).

It applies to the **gross** outcome when there is reinsurance, so the results for `a05g` and `a05no` are the same.

```
In [112]: oco = ['display.width', 150, 'display.max_columns', 15,
   .....:           'display.float_format', lambda x: f'{x:.5g}']
   .....:

In [113]: with pd.option_context(*oco):
   .....:     print(a05g.statistics)
   .....:     print('\n')
   .....:     print(p07.statistics)
   .....:
name                   TenM:05g
component  measure
limit                       inf
attachment                 None
sevcv      param              0
el                           11
prem                          0
lr                            0
freq       ex1                2
           ex2                4
           ex3                8
           mean               2
           cv                 0
           skew             NaN
sev        ex1              5.5
           ex2             38.5
           ex3            302.5
           mean             5.5
           cv           0.52223
           skew               0
agg        ex1               11
           ex2            137.5
           ex3           1875.5
           mean              11
           cv           0.36927
           skew               0
mix        cv             [2.0]
wt                            1


                             A          B        Cat      total
component measure
freq      ex1              100        150          2        252
          ex2            10100      30750          6      71856
          ex3       1.0301e+06 7.9868e+06         22 2.3216e+07
          mean             100        150          2        252
          cv               0.1    0.60553    0.70711    0.36266
          skew             0.1     1.2001    0.70711     1.1783
sev       ex1              100     45.213     616.02     71.484
          ex2            25621     4068.7 4.1047e+06      45166
          ex3        1.674e+07 6.7987e+05 1.7449e+11 1.3919e+09
```

(continues on next page)

(continued from previous page)

```
          mean          100      45.213      616.02      71.484
          cv         1.2498     0.99517      3.1331      2.7998
          skew       5.6619      3.4345      23.278      172.51
agg       ex1         10000      6781.9        1232       18014
          ex2    1.0256e+08  6.3163e+07  9.7273e+06  3.5244e+08
          ex3    1.0785e+12  7.4665e+11  3.8119e+11  7.7915e+12
          mean        10000      6781.9        1232       18014
          cv        0.16006     0.61096      2.3256     0.29343
          skew       0.4082      1.2009      14.837      2.9528
          limit       10000        2500       1e+05       1e+05
          P99.9e      15944       27628       41971       56972
```

The `report_df` Dataframe

The `report_df` dataframe combines information from `statistics` with estimated moments to test if the numerical approximations appear valid. It is an expanded version of `describe`. Numerically estimated statistics are prefaced `est` or `empirical`.

```
In [114]: with pd.option_context(*oco):
   .....:     print(a05g.report_df)
   .....:     print('\n')
   .....:     print(p07.report_df)
   .....:
view                   0 independent     mixed   empirical       error
statistic
name          TenM:05g    TenM:05g  TenM:05g
limit              inf         inf       inf
attachment                       0         0
el                  11          11        11
freq_m               2           2         2
freq_cv              0           0         0
freq_skew
sev_m              5.5         5.5       5.5         5.5           0
sev_cv         0.52223     0.52223   0.52223     0.52223           0
sev_skew             0           0         0           0
agg_m               11          11        11          11 -3.3307e-16
agg_cv         0.36927     0.36927   0.36927     0.36927           0
agg_skew             0           0         0 -6.1064e-14        -inf


unit                   A           B         Cat       total
statistic
freq_m               100         150           2         252
freq_cv              0.1     0.60553     0.70711     0.36266
freq_skew            0.1      1.2001     0.70711      1.1783
sev_m                100      45.213      616.02      71.484
sev_cv            1.2498     0.99517      3.1331      2.7998
sev_skew          5.6619      3.4345      23.278      172.51
agg_m              10000      6781.9        1232       18014
agg_emp_m          10000      6781.8        1232       18013
agg_m_err    -1.1973e-08 -1.2825e-05 -2.7692e-05 -2.6963e-05
agg_cv           0.16006     0.61096      2.3256     0.29343
agg_emp_cv       0.16007     0.61096      2.3249     0.29322
agg_cv_err    6.5168e-06  1.9129e-06 -0.00027305 -0.00070746
agg_skew          0.4082      1.2009      14.837      2.9528
agg_emp_skew     0.40819      1.2009      14.818       2.925
agg_skew_err -1.3542e-05  1.6687e-08  -0.0012766  -0.0094143
agg_emp_kurt     0.40604      2.1621      381.65      33.187
P99.0_emp          14206       19834       10246       33586
P99.6_emp          14956       22626       16610       38304
```

The report_df provides extra information when there is a mixed severity.

```
In [115]: with pd.option_context(*oco):
   .....:     print(a03.report_df)
   .....:
view              0        1        2  independent    mixed  empirical       error
statistic
name         TenM:03  TenM:03  TenM:03      TenM:03  TenM:03
limit            inf      inf      inf          inf      inf
attachment                                        0        0
el              62.5      125    187.5          375      375
freq_m          12.5     6.25     6.25           25       25
freq_cv      0.57446  0.64031  0.64031      0.36572  0.53852
freq_skew     1.0097   1.0307   1.0307      0.66199   1.0028
sev_m              5       20       30           15       15     14.999 -3.7467e-05
sev_cv           0.5    0.375  0.50002      0.90907  0.90907    0.89065   -0.020261
sev_skew           1   2.6716      inf          inf      inf     9.9075          -1
agg_m           62.5      125    187.5          375      375     374.99 -3.8034e-05
agg_cv       0.59161  0.65765  0.67082      0.41265  0.56838     0.5672  -0.0020722
agg_skew      1.0238   1.0613      inf          inf      inf     1.0698          -1
```

The dataframe shows statistics for each mixture component, columns 0,1,2, their sum if they are added independently and their sum if there is a shared mixing variable, as there is here. The common mixing induces correlation between the mix components, acting to increases the CV and skewness, often dramatically.

The spec and spec_ex Dictionaries

The spec dictionary contains the input information needed to create each object. For example, if a is an Aggregate, then Aggregate (**a.spec) creates a new copy. spec_ex appends meta-information to spec about hyper-parameters.

```
In [116]: from pprint import pprint

In [117]: pprint(a05n.spec)
{'agg_kind': 'net of',
 'agg_reins': [(1.0, 4.0, 8.0)],
 'exp_attachment': None,
 'exp_el': 0,
 'exp_en': -1,
 'exp_limit': inf,
 'exp_lr': 0,
 'exp_premium': 0,
 'freq_a': array([2.]),
 'freq_b': array([1.]),
 'freq_name': 'empirical',
 'freq_p0': nan,
 'freq_zm': False,
 'name': 'TenM:05n',
 'note': '',
 'occ_kind': 'net of',
 'occ_reins': [(1.0, 3.0, 7.0)],
 'sev_a': nan,
 'sev_b': 0,
 'sev_conditional': True,
 'sev_cv': 0,
 'sev_lb': 0,
 'sev_loc': 0,
 'sev_mean': 0,
 'sev_name': 'dhistogram',
 'sev_pick_attachments': None,
 'sev_pick_losses': None,
```

(continues on next page)

(continued from previous page)

```
'sev_ps': array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]),
'sev_scale': 0,
'sev_ub': inf,
'sev_wt': 1,
'sev_xs': array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])}
```

The DecL Program

The `program` property returns the DecL program used to create the object. It is blank if the object was not created using DecL. The helper function `pprint_ex()` pretty prints a program.

```
In [118]: from aggregate import pprint_ex

In [119]: pprint_ex(a05n.program, split=20)
Out[119]: 'agg TenM:05n\n  dfreq [2]\n  dsev [1:10]\n  occurrence net of 3 xs 7\n ␣
↪aggregate net of 4 xs 8'

In [120]: pprint_ex(p07.program, split=20)
Out[120]: 'port TenM:07\n  agg A\n    100 claims 10000 xs 0\n    sev lognorm 100␣
↪cv 1.25\n    poisson\n  agg B\n    150 claims 2500 xs 5\n    sev lognorm 50 cv 0.
↪9\n    mixed gamma .6\n  agg Cat\n    2 claims 1e5 xs 0\n    sev 500 * pareto 1.
↪8 - 500\n    poisson'
```

The `update()` Method

After an `Aggregate` or a `Portfolio` object has been created it needs to be updated to populate its `density_df` dataframe. `build()` automatically updates the objects it creates with default hyper-parameter values. Sometimes it is necessary to re-update with different hyper-parameters. The `update()` method takes arguments `log2=13`, `bs=0`, and `recommend_p=0.999`. The first two control the number and size of buckets. When `bs==0` it is estimated using the method `recommend_bucket()`. If `bs!=0` then `recommend_p` is ignored.

Further control over updating is available, as described in REF.

Statistical Functions

`Aggregate` and `Portfolio` objects include basic mean, CV, standard deviation, variance, and skewness statistics as attributes. Those prefixed `agg` are based on exact calculations:

- `agg_m`, `agg_cv`, `agg_sd`, `agg_var`, and `agg_skew`

and prefixed `est` are based on the estimated numerical statistics:

- `est_m`, `est_cv`, `est_sd`, `est_var`, and `est_skew`.

In addition, `Aggregate` has similar series prefixed `sev` and `est_sev` for the exact and estimated numerical severity. These attributes are just conveniences; they are all available in (or derivable from) `report_df`.

`Aggregate` and `Portfolio` objects act like `scipy.stats` (continuous) frozen random variable objects and include the following statistical functions.

- `pmf()` the probability mass function
- `pdf()` the probability density function—broadly interpreted—defined as the pmf divided by `bs`
- `cdf()` the cumulative distribution function
- `sf()` the survival function
- `q()` the quantile function (left inverse cdf), also known as value at risk
- `tvar()` tail value at risk function

- `var_dict()` a dictionary of tail statistics by unit and in total

We aren't picky about whether the density is technically a density when the aggregate is actually mixed or discrete. The discrete output (`density_df.p_*`) is interpreted as the distribution, so none of the statistical functions is interpolated. For example:

```
In [121]: qd(a05g.pmf(2), a05g.pmf(2.2), a05g.pmf(3), a05g.cdf(2), a05g.cdf(2.2))
0.01
0
0.02
0.01
0.01

In [122]: print(1 - a05g.cdf(2), a05g.sf(2))
0.99 0.99

In [123]: print(a05g.q(a05g.cdf(2)))
2.0
```

The last line illustrates that `q()` and `cdf()` are inverses. The `var_dict()` function computes tail statistics for all units, return in a dictionary.

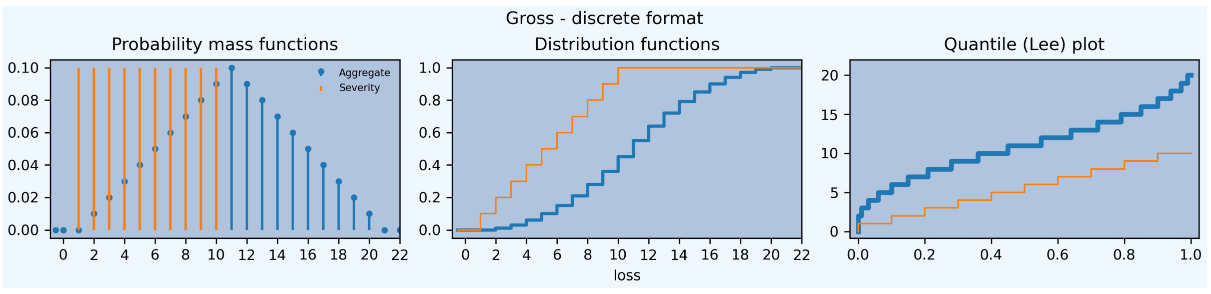
```
In [124]: p07.var_dict(0.99), p07.var_dict(0.99, kind='tvar')
Out[124]:
({'A': 14206.0, 'B': 19834.0, 'Cat': 10246.0, 'total': 33586.0},
 {'A': 15025.781065024481,
  'B': 22845.87765881529,
  'Cat': 20360.45124669143,
  'total': 41763.10848610794})
```

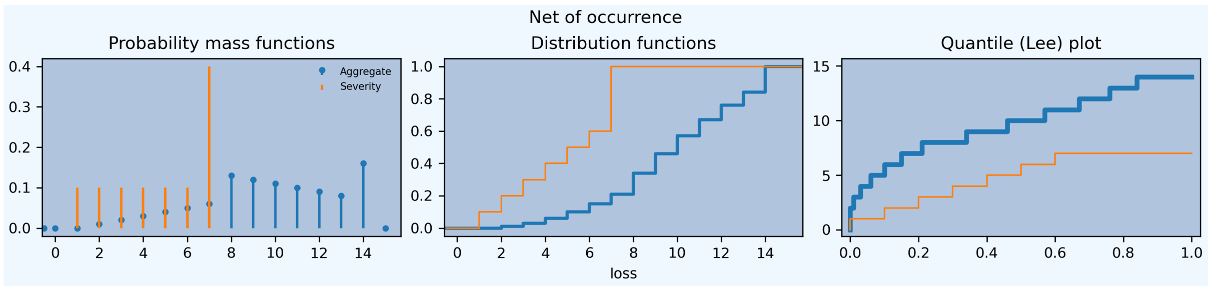
The `plot()` Method

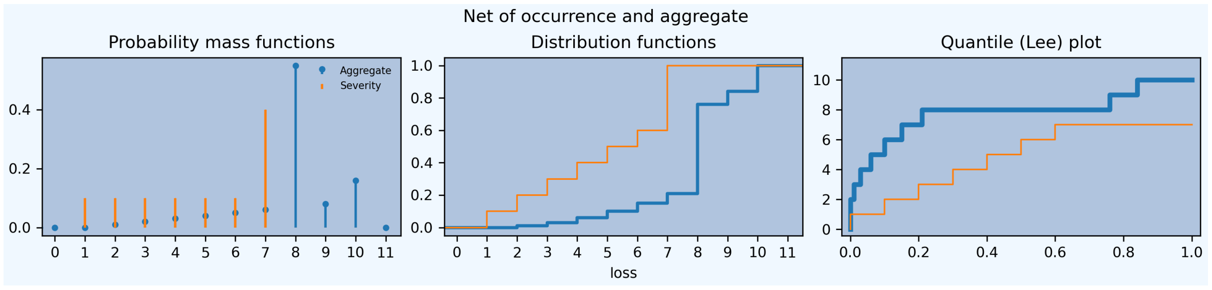
The `plot()` method provides basic visualization. There are three plots: the pdf/pmf for severity and the aggregate on the left. The middle plot shows log density for continuous distributions and the distribution function for discrete ones (selected when `bs==1` and the mean is < 100). The right plot shows the quantile (or VaR or Lee) plot.

The reinsurance examples below show the discrete output format. The plots show the gross, net of occurrence, and net severity and aggregate pmf (left) and cdf (middle), and the quantile (Lee) plot (right). The property `a05g.figure` returns the last figure made by the object as a convenience. You could also use `plt.gcf()`.

```
In [125]: a05g.plot()

In [126]: a05g.figure.suptitle('Gross - discrete format');

In [127]: a05no.plot()

In [128]: a05no.figure.suptitle('Net of occurrence');

In [129]: a05n.plot()

In [130]: a05n.figure.suptitle('Net of occurrence and aggregate');
```

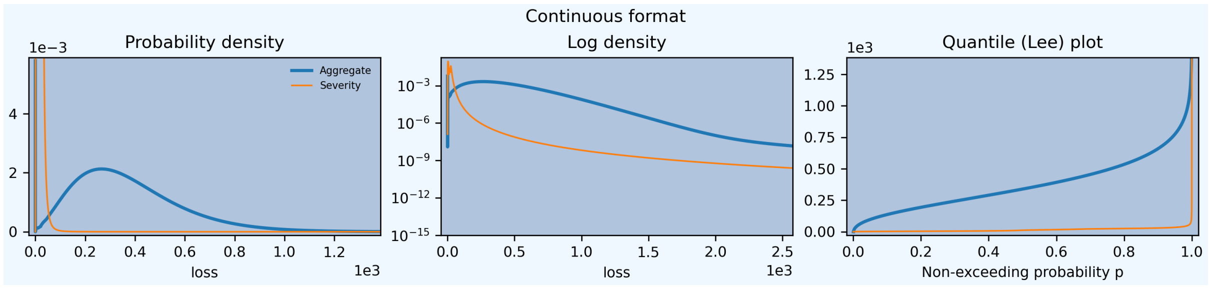






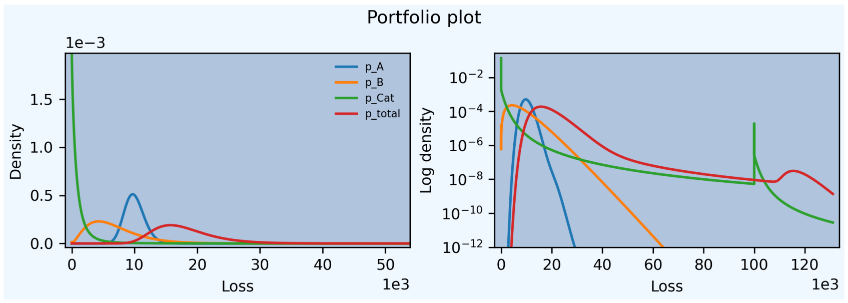
Continuous distributions substitute the log density for the distribution in the middle.

```
In [131]: a03.plot()

In [132]: a03.figure.suptitle('Continuous format');
```



A `Portfolio` object plots the density and log density of each unit and the total.

```
In [133]: p07.plot()

In [134]: p07.figure.suptitle('Portfolio plot');
```



The `price()` Method

The `price()` method computes the risk adjusted expected value (technical price net of expenses) of losses limited by capital at a specified VaR threshold. Suppose the 99.9%ile outcome is used to specify regulatory assets a.

```
In [135]: qd(a03.q(0.999))
1358
```

The risk adjustment is specified by a spectral risk measure corresponding to an input distortion. Distortions can be built using DecL, see *The Distortion Class*. `price()` applies to $X \wedge a$. It returns expected limited losses `L`, the risk adjusted premium `P`, the margin `M = P - L`, the capital `Q = a - P`, the loss ratio, leverage as premium to capital `PQ`, and return on capital `ROE`.

```
In [136]: qd(a03.price(0.999, d06).T)

line        TenM:03
statistic
L            374.82
P            558.25
M            183.43
Q            799.75
a              1358
LR          0.67142
PQ          0.69802
ROE         0.22936
```

When `price()` is applied to a `Portfolio`, it returns the total premium and its (lifted) natural allocation to each unit, see PIR Chapter 14, along with all the other statistics in a dataframe. Losses are allocated by equal priority in default.

```
In [137]: qd(p07.price(0.999, d06).df.T)

distortion Dual Moment, 3.000
unit                        A       B       Cat    total
statistic
L                      9997.3  6779.8    1212.4    17990
P                       10448  9874.5    1913.8    22236
M                      450.25  3094.7    701.35   4246.2
Q                      7656.6   10203     12817    30662
a                       18104   20078     14731    52898
LR                     0.9569 0.68659   0.63352  0.80904
PQ                     1.3645 0.96781   0.14932  0.72519
COC                  0.058806 0.30332  0.054721  0.13848
```

The ROE varies by unit, reflecting different consumption and cost of capital by layer. The less risky unit A runs at a higher loss ratio (cheaper insurance) but higher ROE than unit B because it consumes more expensive, equity-like lower layer capital but less capital overall (higher leverage).

The `snap()` Method

`snap()` rounds an input number to the index of `density_df`. It selects the nearest element.

The `approximate()` Method

The `approximate()` method creates an analytic approximation fit using moment matching. Normal, lognormal, gamma, shifted lognormal, and shifted gamma distributions can be fit, the last two requiring three moments. To fit all five and return a dictionary call with argument `"all"`.

```
In [138]: fzs = a03.approximate('all')

In [139]: d = pd.DataFrame({k: fz.stats('mvs') for k, fz in fzs.items()},
   .....:          index=pd.Index(['mean', 'var', 'skew'], name='stat'),
   .....:                 dtype=float)
   .....:

In [140]: qd(d)

       norm  gamma  lognorm  sgamma  slognorm
stat
mean 374.99 374.99   374.99  374.99    374.99
var   45238  45238    45238   45238     45238
skew      0 1.1344   1.8841  1.0698    1.0698
```

2.3.10 Additional `Portfolio` Methods

Conditional Expected Values

A `Portfolio` object's `density_df` includes a slew of values to allocate capital (please don't) or margin (please do). These all rely on what Mildenhall and Major [2022] call the κ function, defined for a sum $X = \sum_i X_i$ as the conditional expectation

$$\kappa_i(x) = \mathsf{E}[X_i \mid X = x].$$

Notice that $\sum_i \kappa_i(x) = x$, hinting at its allocation application. See PIR Chapter 14.3 for an explanation of why κ is so useful. In short, it shows which units contribute to bad overall outcomes. It is in `density_df` as the columns `exeqa_unit`, read as the "expected value given X eq(uals) a".

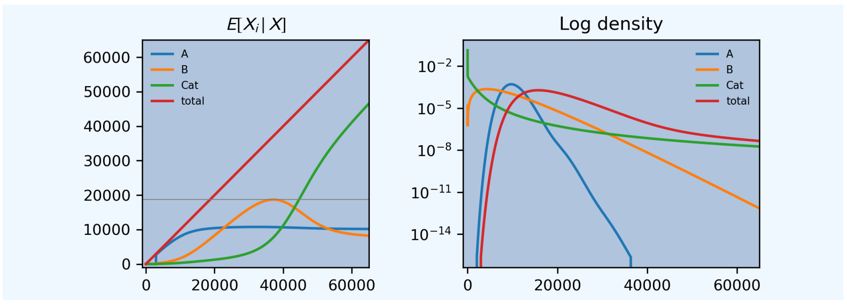
Here are some κ values and graph for `p07`. Looking the log density plot on the right shows that unit B dominates for moderately large events, but Cat dominates for the largest events.

```
In [141]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45)); \
   .....: ax0, ax1 = axs.flat; \
   .....: lm = [-1000, 65000]; \
   .....: bit = p07.density_df.filter(regex='exeqa_[ABCt]').rename(
   .....:     columns=lambda x: x.replace('exeqa_', '')).sort_index(axis=1); \
   .....: bit.index.name = 'Loss'; \
   .....: bit.plot(xlim=lm, ylim=lm, ax=ax0); \
   .....: ax0.set(title=r'$E[X_i\mid X]$', aspect='equal'); \
   .....: ax0.axhline(bit['B'].max(), lw=.5, c='C7');
   .....:

In [142]: p07.density_df.filter(regex='p_[ABCt]').rename(
   .....:     columns=lambda x: x.replace('p_', '')).plot(ax=ax1, xlim=lm,␣
↪logy=True);
   .....:

In [143]: ax1.set(title='Log density');
```

(continues on next page)

(continued from previous page)

```
In [144]: bit['Pct A'] = bit['A'] / bit.index

In [145]: qd(bit.loc[:lm[1]:1024])

               A       B     Cat   total    Pct A
Loss
0.0            0       0       0       0      NaN
2048.0         0       0       0    2048        0
4096.0    3729.8 307.48 58.741    4096 0.91059
6144.0    5419.3 602.16 122.54    6144 0.88205
8192.0    6911.6 1062.4 217.99    8192  0.8437
10240.0   8133.9 1757.5 348.62   10240 0.79432
12288.0   9041.7 2735.6 510.69   12288 0.73582
14336.0   9655.4 3986.6 694.09   14336  0.6735
16384.0    10048 5445.7 890.17   16384 0.61329
18432.0    10299 7035.3   1098   18432 0.55874
20480.0    10464 8692.1 1324.3   20480 0.51092
22528.0    10576  10370 1582.2   22528 0.46946
...          ...    ...    ...     ...      ...
40960.0    10622  17734  12604   40960 0.25932
43008.0    10551  16531  15926   43008 0.24532
45056.0    10477  15060  19519   45056 0.23253
47104.0    10408  13547  23149   47104 0.22096
49152.0    10349  12178  26625   49152 0.21055
51200.0    10301  11051  29847   51200 0.20119
53248.0    10264  10185  32799   53248 0.19276
55296.0    10236 9545.6  35515   55296 0.18511
57344.0    10214 9085.2  38045   57344 0.17812
59392.0    10197 8755.1  40440   59392 0.17169
61440.0    10184   8516  42740   61440 0.16575
63488.0    10172 8339.2  44976   63488 0.16023
```



The thin horizontal line at the maximum value of `exeqa_B` (left plot) shows that κ_B is not increasing. Unit B contributes more to moderately bad outcomes than Cat, but in the tail Cat dominates.

Using `filter(regex=...)` to select columns from `density_df` is a helpful idiom. The total column is labeled `_total`. Using upper case for unit names makes them easier to select.

Calibrate Distortions

The `calibrate_distortions()` method calibrates distortions to achieve requested pricing for the total loss. Pricing can be requested by loss ratio or return on capital (ROE). Asset levels can be specified in monetary terms, or as a probability of non-exceedance. To calibrate the usual suspects (constant cost of capital, proportional hazard, dual, Wang, and TVaR) to achieve a 15% return with a 99.6% capital level run:

```
In [146]: p07.calibrate_distortions(Ps=[0.996], ROEs=[0.15], strict='ordered');

In [147]: qd(p07.distortion_df)

                                  S     L     P     PQ     Q  COC   param      error
a        LR       method
38304.0 0.871293 ccoc   0.0039991 17962 20615 1.1654 17689 0.15    0.15          0
                 ph     0.0039991 17962 20615 1.1654 17689 0.15 0.61919  1.095e-09
                 wang   0.0039991 17962 20615 1.1654 17689 0.15 0.51526 4.7105e-06
                 dual   0.0039991 17962 20615 1.1654 17689 0.15  2.0031 -2.8012e-09
                 tvar   0.0039991 17962 20615 1.1654 17689 0.15 0.37271 9.7863e-06

In [148]: pprint(p07.dists)
{'ccoc': ccoc (0.14999999999999997, 0),
 'dual': dual (2.003099750057873),
 'ph': ph (0.6191915749880643),
 'tvar': tvar (0.37271484907821667),
 'wang': wang (0.5152633577003828)}
```

The answer is returned in the `dist_ans` dataframe. The requested distortions are all single parameter, returned in the `param` column. The last column gives the error in achieved premium. The attribute `p07.dists` is a dictionary with keys distortion types and values `Distortion` objects. See PIR REF for more discussion.

Analyze Distortions

The `analyze_distortions()` method applies the distortions in `p07.dists` at a given capital level and summarizes the implied (lifted) natural allocations across units. Optionally, it applies a number of traditional (bullshit) pricing methods. The answer dataframe includes premium, margin, expected loss, return, loss ratio and leverage statistics for each unit and method. Here is a snippet, again at the 99.6% capital level.

```
In [149]: ans = p07.analyze_distortions(p=0.996)

In [150]: print(ans.comp_df.xs('LR', axis=0, level=1).
   .....:       to_string(float_format=lambda x: f'{x:.1%}'))
   .....:
line             A      B   Cat  total
Method
Dist ccoc   109.8% 103.8% 24.0%  87.1%
Dist dual    96.9%  77.8% 75.4%  87.1%
Dist ph      99.0%  80.6% 56.4%  87.1%
Dist tvar    96.3%  77.8% 77.8%  87.1%
Dist wang    97.7%  78.6% 67.7%  87.1%
EL           87.1%  87.1% 87.1%  87.1%
EPD          96.0%  78.3% 10.3%  58.6%
EqRiskEPD    99.7%  90.7% 38.4%  87.1%
EqRiskTVaR   95.7%  83.5% 58.0%  87.1%
EqRiskVaR    95.5%  82.7% 61.0%  87.1%
MerPer       97.9%  88.9% 66.0%  91.5%
ScaledEPD   107.9%  99.5% 26.3%  87.1%
ScaledTVaR   99.0%  85.3% 46.3%  87.1%
ScaledVaR    99.3%  84.9% 46.5%  87.1%
TVaR         94.3%  78.0% 38.0%  80.1%
VaR          93.9%  76.6% 37.3%  79.2%
```

(continues on next page)

(continued from previous page)

```
coTVaR       99.1%  83.5% 49.3%  87.1%
covar        97.6%  80.6% 60.5%  87.1%
```

Twelve Plot

The `twelve_plot()` method produces a detailed analysis of the behavior of a two unit portfolio. To run it, build the portfolio and calibrate some distortions. Then apply one of the distortions (to compute an augmented version of `density_df` with pricing information). We give two examples.

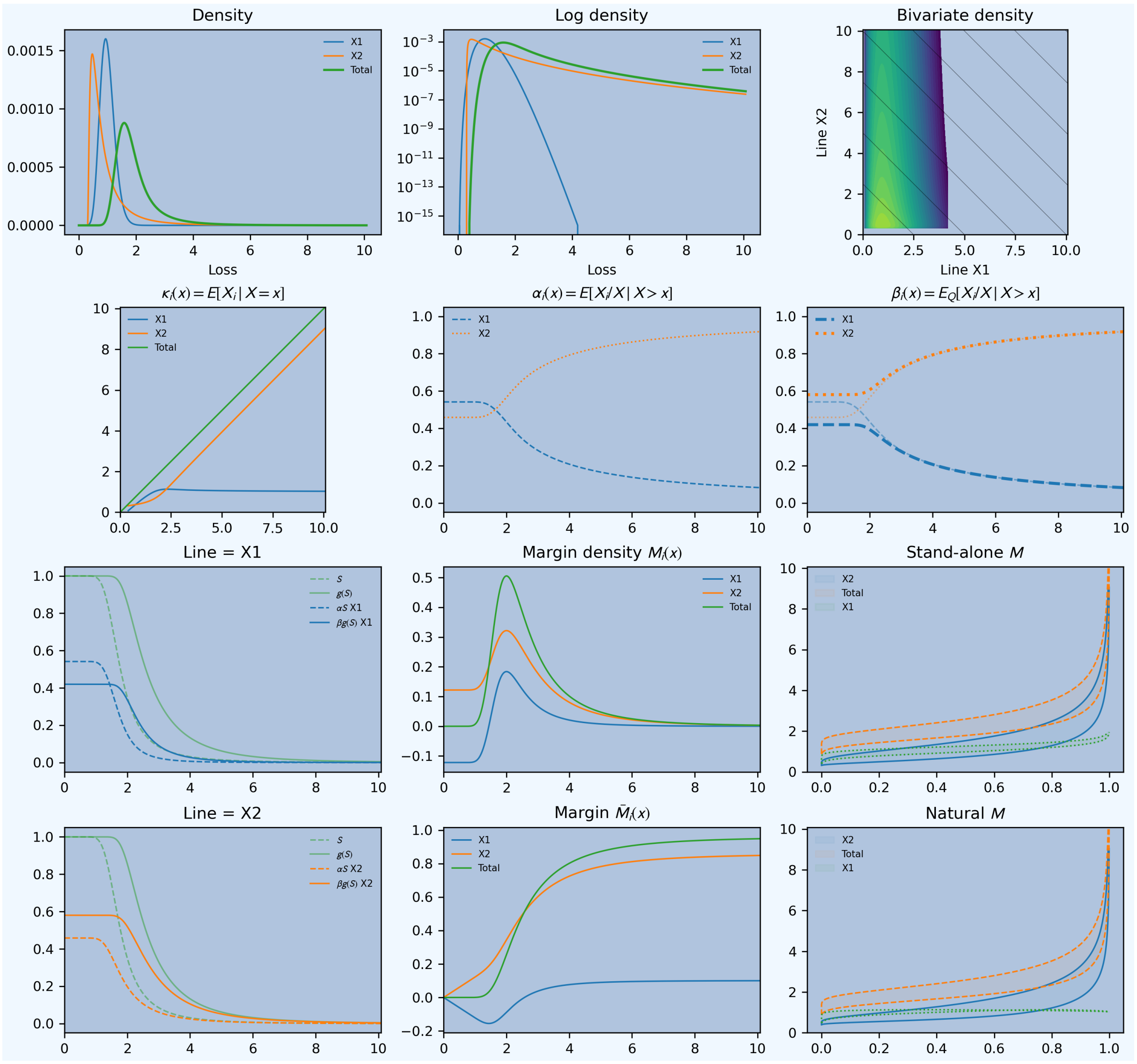
First, the case of a thin-tailed and a thick-tailed unit. Here, the thick tailed line benefits from pooling at low capital levels, resulting in negative margins to the thin-tail line in compensation. At moderate to high capital levels the total margin for both lines is positive. Assets are 12.5. The argument `efficient=False` in `apply_distortion()` includes extra columns in `density_df` that are needed to compute the plot.

```
In [151]: p09 = build('port TenM:09 '
   .....:              'agg X1 1 claim sev gamma 1 cv 0.25 fixed '
   .....:              'agg X2 1 claim sev 0.7 * lognorm 1 cv 1.25 + 0.3 fixed'
   .....:             , bs=1/1024)
   .....:

In [152]: qd(p09)

            E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
unit  X
X1    Freq     1                            0
      Sev      1        1 -5.5511e-16    0.25      0.25     0.5         0.5
      Agg      1        1 -5.5511e-16    0.25      0.25     0.5         0.5
X2    Freq     1                            0
      Sev      1  0.99999 -1.0796e-05   0.875   0.87452  5.7031       5.603
      Agg      1  0.99999 -1.0796e-05   0.875   0.87452  5.7031       5.603
total Freq     2                            0
      Sev      1  0.99999  -5.398e-06 0.64348            7.1845
      Agg      2        2 -5.8107e-06 0.45501   0.45476  5.0802      4.9869
log2 = 16, bandwidth = 1/1024, validation: fails sev skew, agg skew.

In [153]: print(f'Asset P value {p09.cdf(12.5):.5g}')
Asset P value 0.99958

In [154]: p09.calibrate_distortions(ROEs=[0.1], As=[12.5], strict='ordered');

In [155]: qd(p09.distortion_df)

                              S      L      P      PQ      Q  COC   param      ␣
↪error
a    LR       method                                                            ␣
↪
12.5 0.676722 ccoc   0.00041558 1.9985 2.9531 0.30933 9.5469  0.1      0.1      ␣
↪ 0
              ph     0.00041558 1.9985 2.9531 0.30933 9.5469  0.1 0.51152   1.998e-
↪08
              wang   0.00041558 1.9985 2.9531 0.30933 9.5469  0.1 0.88394  1.6577e-
↪06
              dual   0.00041558 1.9985 2.9531 0.30933 9.5469  0.1  4.4902 -2.0466e-
↪10
              tvar   0.00041558 1.9985 2.9531 0.30933 9.5469  0.1 0.71207  6.2397e-
↪06

In [156]: p09.apply_distortion('dual', efficient=False);
```

(continues on next page)

(continued from previous page)

```
In [157]: fig, axs = plt.subplots(4, 3, figsize=(3 * 3.5, 4 * 2.45), constrained_
↪layout=True)

In [158]: p09.twelve_plot(fig, axs, p=0.999, p2=0.999)
```



There is a lot of information here. We refer to the charts as (r, c) for row $r = 1, 2, 3, 4$ and column $c = 1, 2, 3$, starting at the top left. The horizontal axis shows the asset level in all charts except $(3, 3)$ and $(4, 3)$, where it shows probability, and $(1, 3)$ where it shows loss. Blue represents the thin tailed unit, orange thick tailed and green total. When both dashed and solid lines appear on the same plot, the solid represent risk-adjusted and dashed represent non-risk-adjusted functions. Here is the key.

- $(1, 1)$ shows density for X_1, X_2 and $X = X_1 + X_2$; the two units are independent. Both units have mean 1.
- $(1, 2)$: log density; comparing tail thickness.
- $(1, 3)$: the bivariate log-density. This plot illustrates where (X_1, X_2) *lives*. The diagonal lines show $X = k$ for different k. These show that large values of X correspond to large values of X_2, with X_1 about average.
- $(2, 1)$: the form of κ_i is clear from looking at $(1, 3)$. κ_1 peaks above 1.0 around $x = 2$ and hereafter it declines to 1.0. κ_2 is monotonically increasing.
- $(2, 2)$: The α_i functions. For small x the expected proportion of losses is approximately 50/50, since the means are equal. As x increases X_2 dominates. The two functions sum to 1.

- $(2,3)$: The thicker lines are β_i and the thinner lines α_i from $(2,2)$. Since α_1 decreases $\beta_1(x) \le \alpha_1(x)$. This can lead to X_1 having a negative margin in low asset layers. X_2 is the opposite.
- $(3,1)$: illustrates premium and margin determination by asset layer for X_1. For low asset layers $\alpha_1(x)S(x) > \beta_1(x)g(S(x))$ (dashed above solid) corresponding to a negative margin. Beyond about $x = 1.38$ the lines cross and the margin is positive.
- $(4,1)$: shows the same thing for X_2. Since α_2 is increasing, $\beta_2(x) > \alpha_2(x)$ for all x and so all layers get a positive margin. The solid line $\beta_2 gS$ is above the dashed $\alpha_2 S$ line.
- $(3,2)$: the layer margin densities. For low asset layers premium is fully funded by loss with zero overall margin. X_2 requires a positive margin and X_1 a negative one, reflecting the benefit the thick unit receives from pooling in low layers. The overall margin is always non-negative. Beyond about $x = 1.5$, X_1's margin is also positive.
- $(4,2)$: the cumulative margin in premium by asset level. Total margin is zero in low *dollar-swapping* layers and then increases. It is always non-negative. The curves in this plot are the integrals of those in $(3,2)$ from 0 to x.
- $(3,3)$: shows stand-alone loss $(1-S(x),x) = (p,q(p))$ (dashed) and premium $(1-g(S(x)),x) = (p,q(1-g^{-1}(1-p))$ (solid, shifted left) for each unit and total. The margin is the shaded area between the two. Each set of three lines (solid or dashed) does not add up vertically because of diversification. The same distortion g is applied to each unit to the stand-alone S_{X_i}. It is calibrated to produce a 10 percent return overall. On a stand-alone basis, calculating capital by unit to the same return period as total, X_1 is priced to a 77.7 percent loss ratio and X_2 52.5 percent, producing an average 62.7 percent, vs. 67.6 percent on a combined basis. Returns are 37.5 percent and 9.4 percent respectively, averaging 11.5 percent, vs 10 percent on a combined basis, see stand-alone analysis below.
- $(4,3)$: shows the natural allocation of loss and premium to each unit. The total (green) is the same as $(3,3)$. For each i, dashed shows $(p, \mathsf{E}[X_i \mid X = q(p)])$, i.e. the expected loss recovery conditioned on total losses $X = q(p)$, and solid shows $(p, \mathsf{E}[X_i \mid X = q(1-g^{-1}(1-p))])$, i.e. the natural premium allocation. Here the solid and dashed lines *add up* vertically: they are allocations of the total. Looking vertically above p, the shaded areas show how the total margin at that loss level is allocated between lines. X_1 mostly consumes assets at low layers, and the blue area is thicker for small p, corresponding to smaller total losses. For p close to 1, large total losses, margin is dominated by X_2 and in fact X_1 gets a slight credit (dashed above solid). The change in shape of the shaded margin area for X_1 is particularly evident: it shows X_1 benefits from pooling and requires a lower overall margin.

There may appear to be a contradiction between figures $(3,2)$ and $(4,3)$ but it should be noted that a particular p value in $(4,3)$ refers to different events on the dotted and solid lines.

Plots $(3,3)$ and $(4,3)$ explain why the thick unit requires relatively more margin: its shape does not change when it is pooled with X_1. In $(3,3)$ the green shaded area is essentially an upwards shift of the orange, and the orange areas in $(3,3)$ and $(3,4)$ are essentially the same. This means that adding X_1 has virtually no impact on the shape of X_2; it is like adding a constant. This can also be seen in $(4,3)$ where the blue region is almost a straight line.

Applying the same distortion on a stand-alone basis produces:

```
In [159]: a = p09.stand_alone_pricing(p09.dists['dual'], p=p09.cdf(12.5))

In [160]: print(a.iloc[:8])
line                                  X1          X2          sop       total
method                      stat
sa Dual Moment, 4.490  L      0.999956    0.998459    1.998414    1.998459
                       LR     0.777176    0.525226    0.626922    0.676722
                       M      0.286697    0.902547    1.189244    0.954686
                       P      1.286652    1.901006    3.187658    2.953145
                       PQ     1.681666    0.198284    0.307915    0.309332
                       Q      0.765106    9.587275   10.352381    9.546855
                       ROE    0.374715    0.094140    0.114876    0.100000
                       a      2.051758   11.488281   13.540039   12.500000
```

The lifted natural allocation (diversified pricing) is given next. These numbers are so different than the stand-alone because X2 has to compensate X1 for the transfer of wealth in default states. When there is a large loss, it is caused by X2 and so X2 receives a disproportionate share of the assets in default.

```
In [161]: a2 = p09.analyze_distortion('dual', ROE=0.1, p=p09.cdf(12.5))

In [162]: print(a2.pricing.unstack(1).droplevel(0, axis=0).T)
line            X1        X2     total
     stat
12.5 L     0.999920  0.998539  1.998459
     LR    0.908669  0.538958  0.676722
     M     0.100502  0.854183  0.954686
     P     1.100422  1.852722  2.953145
     PQ    0.888570  0.222992  0.309332
     Q     1.238419  8.308458  9.546855
     ROE   0.081154  0.102809  0.100000
```

The second portfolio has been selected with two thick tailed units. A appears riskier at lower return periods and B at higher. Pricing is calibrated to a 15% ROE at a 99.6% capital level.

```
In [163]: p10 = build('port TenM:10 '
   .....:                 'agg A '
   .....:                     '30 claims '
   .....:                     '1000 xs 0 '
   .....:                     'sev gamma 25 cv 1.5 '
   .....:                     'mixed delaporte 0.75 0.6 '
   .....:                 'agg B '
   .....:                     '5 claims '
   .....:                     '20000 xs 20 '
   .....:                     'sev lognorm 25 cv 3.0 '
   .....:                     'poisson'
   .....:               , bs=1)
   .....:

In [164]: qd(p10)

             E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)   Skew(X) Est Skew(X)
unit  X
A     Freq     30                      0.7719              3.5645
      Sev      25   24.989 -0.00045632     1.5     1.501         3      2.9984
      Agg     750   749.66 -0.00045632 0.81904   0.81911    3.2482      3.2478
B     Freq      5                     0.44721             0.44721
      Sev  55.582    55.58 -3.0222e-05  2.3472    2.3473    18.671      18.671
      Agg  277.91    277.9 -3.0222e-05   1.141     1.141    6.9737      6.9737
total Freq     35                     0.66471              3.5156
      Sev  29.369   29.359 -0.00034112  2.0855              27.142
      Agg  1027.9   1027.6 -0.00034112 0.67253   0.67256    2.9521      2.9519
log2 = 16, bandwidth = 1, validation: fails sev mean, agg mean.

In [165]: p10.calibrate_distortions(ROEs=[0.15], Ps=[0.996], strict='ordered');

In [166]: qd(p10.distortion_df)

                               S      L      P      PQ      Q   COC   param     ␣
↪error
a      LR      method                                                          ␣
↪
4596.0 0.68709 ccoc   0.0039969 1023.3 1489.3 0.47937 3106.7 0.15     0.15     ␣
↪0
               ph     0.0039969 1023.3 1489.3 0.47937 3106.7 0.15 0.57953 3.9482e-
↪06
               wang   0.0039969 1023.3 1489.3 0.47937 3106.7 0.15 0.64115  3.716e-
↪06
               dual   0.0039969 1023.3 1489.3 0.47937 3106.7 0.15  2.6459      ␣
↪0
               tvar   0.0039969 1023.3 1489.3 0.47937 3106.7 0.15 0.52695 8.0193e-
```

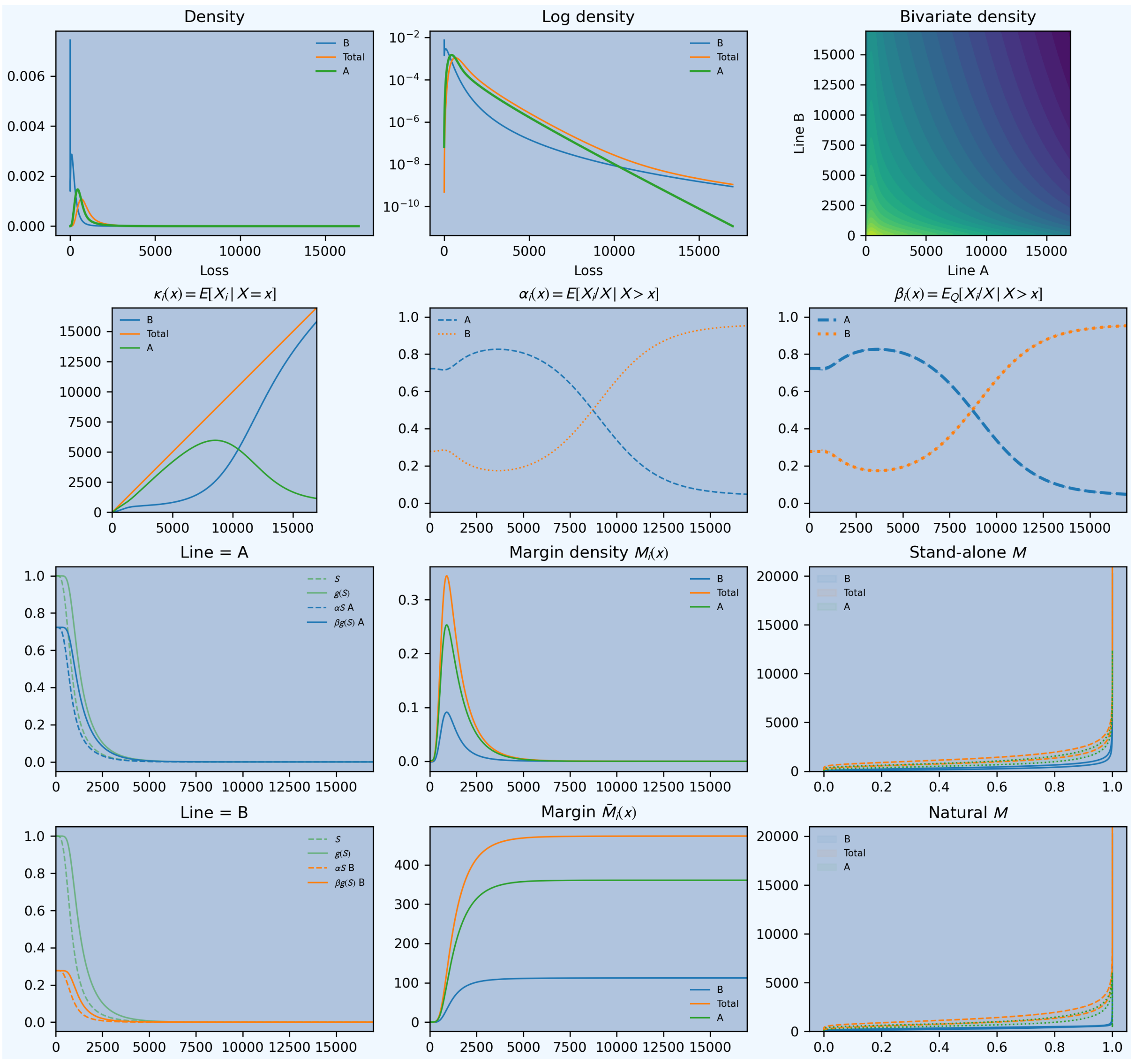
(continues on next page)

(continued from previous page)

```
↪06
```

Apply the dual distortion and then create the twelve plot.

```
In [167]: p10.apply_distortion('dual', efficient=False);

In [168]: fig, axs = plt.subplots(4, 3, figsize=(3 * 3.5, 4 * 2.45), constrained_
↪layout=True)

In [169]: p10.twelve_plot(fig, axs, p=0.999995, p2=0.999999)
```



Applying the same distortion on a stand-alone basis produces:

```
In [170]: assets = p10.q(0.996)

In [171]: a = p10.stand_alone_pricing(p10.dists['dual'], p=p10.cdf(assets))

In [172]: print(a.iloc[:8])
line                                 A            B           sop         total
method                      stat
sa Dual Moment, 2.646 L       745.845783   274.056275   1019.902058   1023.266475
                      LR        0.652332     0.590524      0.634487      0.687090
```

(continues on next page)

(continued from previous page)

```
                  M      397.506631   190.033998   587.540629   466.008721
                  P     1143.352413   464.090274  1607.442687  1489.275195
                  PQ       0.383850     0.309206     0.358840     0.479371
                  Q     2978.647587  1500.909726  4479.557313  3106.724805
                  ROE      0.133452     0.126613     0.131160     0.150000
                  a     4122.000000  1965.000000  6087.000000  4596.000000
```

The lifted natural allocation (diversified pricing) is given next.

```
In [173]: a2 = p10.analyze_distortion('dual', ROE=0.1, p=p10.cdf(assets))

In [174]: print(a2.pricing.unstack(1).droplevel(0, axis=0).T)
line                    A            B         total
       stat
4596.0 L       746.454112  276.812363  1023.266475
       LR        0.752826    0.776418     0.759066
       M       245.081324   79.712633   324.793957
       P       991.535436  356.524996  1348.060432
       PQ        0.379589    0.560741     0.415051
       Q      2612.128741  635.810828  3247.939568
       ROE       0.093824    0.125372     0.100000
```

2.3.11 Extensions

The `extensions` sub-package contains additional classes and functions that are either peripheral to the main project or still under development (and subject to change). Currently, `extensions` includes:

- `case_studies` for creating and managing PIR case studies (see *Case Studies*).
- `pir_figures` for creating various exhibits and figures in PIR.
- `figures` for creating various other exhibits and figures.
- `samples` includes functions for working with samples and executing a switcheroo. Eventually, these will be integrated into `Portfolio`.

2.3.12 Summary of Objects Created by DecL

Each of the objects created by `build()` is automatically stored in the knowledge. We can list them out now.

```
In [175]: from aggregate import pprint_ex

In [176]: for n, r in build.qlist('^TenM:').iterrows():
   .....:     pprint_ex(r.program, split=20)
   .....:
```

2.4 The Dec Language

Objectives: Introduce the Dec Language (DecL) used to specify aggregate distributions in familiar insurance terminology.

Audience: User who wants to use DecL to build realistic aggregates.

Prerequisites: Familiar with using `build`. Probability theory behind aggregate distributions. Insurance and reinsurance terminology.

See also: *Reinsurance Pricing*, and *Dec Language Reference*.

Notation: `<item>` denotes an optional term. See the note 10 mins formatting for important information about how DecL programs are formatted and laid out in the help.

Contents:

2.4.1 DecL Design and Purpose

The Dec Language, or simply DecL, is designed to make it easy to go from "Dec page to distribution" — hence the name. An insurance policy's Declarations page spells out key coverage terms and conditions such as the limit and deductible, effective date, named insured, and covered property. A reinsurance slip performs the same functions.

Coverage expressed concisely in words on a Dec page is often incomplete and hard to program. Consider the declaration

> "Aggregate losses from trucking policy with a premium of 2000, a limit of 1000, and no deductible."

To estimate the distribution of outcomes for this policy, the actuary must:

1. Estimate the priced loss ratio on the policy to determine the loss pick (expected loss) as premium times loss ratio. Say they select 67.5%.
2. Select a suitable trucking ground-up severity curve, say lognormal with mean 100 and CV 1.75.
3. Compute the expected conditional layer severity for the layer 1000 xs 0.
4. Divide severity into the loss pick to determine the expected claim count.
5. Select a suitable frequency distribution, say Poisson.
6. Calculate a numerical approximation to the resulting compound-Poisson aggregate distribution

A DecL program takes care of many of these details. The DecL program corresponding to the trucking policy is simply:

```
agg Trucking                      \
    2000 premium at 0.675 lr      \
    1000 xs 0                     \
    sev lognorm 100 cv 1.75       \
    poisson
```

It specifies the loss ratio and distributions selected in steps 1, 2 and 5; these require actuarial judgment and cannot be automated. Based on this input, the `aggregate` package computes the rest of steps 1, 3, 4, and 6. The details of the program are explained in the rest of this chapter.

Note: All DecL programs are one-line long. The program above uses a Python \ line break so that the code above can be cut and pasted as an argument to `build` using a triple quoted string. See 10 mins formatting.

Specifying a Realistic Aggregate Distribution

The trucking example hints at the complexity of specifying a realistic insurance aggregate distribution. Abstracting the details, a complete specification has seven parts:

1. A name
2. The exposure, optionally including occurrence limits and deductibles
3. The ground-up severity distribution
4. Occurrence reinsurance (optional)
5. The frequency distribution
6. Aggregate reinsurance (optional)

7. Additional notes (optional)

DecL follows the same pattern:

```
agg name                    \
    exposure <limit>        \
    severity                \
    <occurrence re>         \
    <frequency>             \
    <aggregate re>          \
    <note>
```

where `<...>` denotes an optional clause. All programs are one-line long and horizontal white space is ignored.

DecL programs are built (interpreted) using the `build` function. Python automatically concatenates strings between parenthesis (no need for `\`), making it is easiest and clearest to enter a program as:

```
build('agg Trucking '
      '2000 premium at 0.675 lr '
      '1000 xs 0 '
      'sev lognorm 100 cv 1.75 '
      'poisson')
```

The entries in this example are as follows.

- `agg` is the DecL keyword used to create an aggregate distribution. Keywords are part of the language, like `if/then/else` in VBA, R or Python, or `select` in SQL.
- `Trucking` is a string name. It can contain letters and numbers and periods and must start with a letter. It is case sensitive. It cannot contain an underscore. It cannot be a DecL keyword. E.g., `Motor`, `NE.Region`, `Unit.A` but not `12Line` or `NE_Region`.
- The exposure clause is `2000 premium at 0.675 lr 1000 xs 0`. (Percent notation is acceptable: the loss ratio can be entered as `67.5% lr`.) It determines the volume of insurance, see *The Exposure Clause*. It includes `1000 xs 0`, an optional *layers subclause* to set policy occurrence limits and deductibles.
- The severity clause `sev lognorm 100 cv 1.75` determines the **ground-up** severity, see *severity*. `sev` is a keyword
- The `frequency` clause, `poisson`, specifies the frequency distribution, see *frequency*.

The occurrence re, aggregate re and note clauses are omitted. See 2_agg_class_reinsurance_clause and *The Note Clause*.

`aggregate` automatically computes the expected claim count from the premium, expected loss ratio, and average severity.

Python `f`-strings allow variables to be passed into DecL programs, `f'sev lognorm {x} cv {cv}`.

Alternative Specifications

There are two other specifications for different situations that reference a distribution from the `knowledge` database.

The first simply refers to the object by name, prefixing it with `agg.`. Thus:

```
agg.Trucking
```

refers to the `Trucking` example above.

The second allows the flexibility to provide a new name for the object:

```
agg NewTruckingAccount agg.Trucking
```

These forms are mostly used in portfolios. See the *Dec Language Reference*.

The rest of this Chapter describes the basic features of each clause.

2.4.2 The Exposure Clause

The exposure clause has two parts: exposures and an optional layers sub-clause described in *The Limits Sub-Clause*. It specifies the volume of insurance. There are five forms:

1. Expected loss
2. Premium and loss ratio
3. Exposure and rate
4. Claim count
5. Using the `dfreq` keyword to enter the frequency distribution directly

Examples:

```
1000 loss
1000 premium at 0.7 lr
5 exposure at 2000 rate
123 claims
dfreq [1 2 3] [3/4 3/16 1/16]
```

- `1000 loss` directly specifies expected loss. The claim count is derived from average severity. It is typical for an actuary to estimate the loss pick and select a severity curve, and then derive frequency.
- `1000 premium at 0.7 lr` directly specifies premium and a loss ratio. Expected losses equal the product. The claim count is again derived from severity. Actuaries often take plan premiums and apply loss ratio picks to determine losses, rather than starting with a loss pick. This idiom supports that approach.
- `5 exposure at 2000 rate` directly specifies exposure and a loss rate. It is analogous to the loss ratio form. Actuaries often know exposure and unit rates (per vehicle, per 100 insured value, per location). This idiom supports that approach.
- `123 claims` directly specifies the expected claim count; the last letter `s` on `claims` is optional, allowing `1 claim`. Expected losses equal claim count times average severity.
- `dfreq [1 2 3] [3/4 3/16 1/16]` specifies frequency outcomes and probabilities directly. It is described in *Non-Parametric Frequency Distributions*.

All values in the first three specifications can be vectorized, see *Vectorization: Limit Profiles and Mixed Severity*.

Determining Expected Claim Count

Variables are used in the following order to determine overall expected losses.

- If `count` is given it is used and loss is derived from severity.
- Else if `loss` is given, then count is derived from the severity.
- Else if either `pp premium at xx lr` or `ee exposure at rr rate` is given, then the loss is derived by multiplication and counts from severity.
- In all cases, if `premium` is given the loss ratio is computed

These choices present no ambiguity when using DecL. But the input arguments could conflict if you create the object directly.

By default, claim count is conditional on a loss to the layer, but severity can have a mass at zero. The severity can be specified to be unconditional, see *Unconditional Severity*.

Details.

In terms of `exp_en`, `exp_el`, `exp_premium`, and `exp_lr` the second and third steps are:

```
exp_el = np.where(exp_el > 0, exp_el, exp_premium * exp_lr)
```

i.e., expected losses are used if given and premium times loss ratio used if not. All these values default to 0. At this point the object must know either loss or claim count:

```
assert np.all( exp_el > 0 or exp_en > 0 )
```

Then

- If `exp_en` is input, it determines the expected claim count; expected losses determined from expected severity
- Else if `exp_el > 0` then it is used as expected loss and claim count determined from severity

Finally,

- If `exp_prem > 0` then the the loss ratio is computed
- Else if `exp_lr > 0` the premium is computed

Thus, if only `exp_en` or `exp_loss` is entered, the object knows loss, but not premium or loss ratio.

2.4.3 The Limits Sub-Clause

The optional `limits` sub-clause specifies policy occurrence limits and deductibles.

Examples:

```
100 xs 0
inf xs 100
750 xs 250
1 xs 1
```

- `100 xs 0` applies an occurrence limit of 100.
- `inf xs 100` applies a deductible of 100 and no limit.
- `750 xs 250` is an excess layer, with limit 750 and deductible 250.
- `1 xs 1` is also an excess layer of 1 xs 1.

`inf` denotes infinity, for an unlimited layer. Both `xs` and `x` are acceptable.

Multiple layers can be entered at once using vectors.

2.4.4 The Severity Clause

The severity clause specifies the ground-up severity distribution, or “severity curve” as it is sometimes known. It is a very flexible clause. Its design follows the `scipy.stats` package’s specification of random variables using shape, location, and scale factors, see *probability background*. The syntax is different for non-parametric discrete distributions and parametric continuous distributions.

Non-Parametric Severity Distributions

Discrete distributions (supported on a finite number of outcomes) can be directly entered as a severity using the `dsev` keyword followed by two equal-length row vectors. The first gives the outcomes and the (optional) second gives the probabilities.

```
dsev [outcomes] <[probabilities]>
```

The horizontal layout is irrelevant and commas are optional. If the `probabilities` vector is omitted then all probabilities are set equal to the reciprocal of the length of the `outcomes` vector. A Python-like colon notation is available for ranges. Probabilities can be entered as fractions, but no other arithmetic operation is supported.

Examples:

```
dsev [0 9 10] [0.5 0.3 0.2]
dsev [0 9 10]
dsev [1:6]
dsev [0:100:25]
dsev [1:6] [1/4 1/4 1/8 1/8 1/8 1/8]
```

- `dsev [0 9 10] [0.5 0.3 0.2]` is a severity with a 0.5 chance of taking the value 0, 0.3 chance of 9, and 0.2 of 10.
- `dsev [0 9 10]` gives equally likely outcomes of 0, 9, or 10.
- `dsev [1:6]` gives equally likely outcomes 1, 2, 3, 4, 5, 6. Unlike Python (but like `pandas.DataFrame.loc`) the right-hand limit is included.
- `dsev [0:100:25]` gives qually likely outcomes 0, 25, 50, 100.
- `dsev [1:6] [1/4 1/4 1/8 1/8 1/8 1/8]` gives outcomes 1 or 2 with probability 0.25 or 3-6 with probability 0.125.

Warning: Use binary fractions (denominator a power of two) to avoid rounding errors!

Details

A `dsev` clause is converted by the parser into a `dhistogram` step distribution:

```
sev dhistogram xps [outcomes] [probabilities]
```

In rare cases you want a continuous (ogive, piecewise linear distribution) version:

```
sev chistogram xps [outcomes] [probabilities]
```

When executed, these are both converted into a `scipy.stats` `histogram` class.

Discrete severities, specified using the `dsev` keyword, are implemented using a `scipy.stats` `rv_historgram` object, which is actually continuous. They work by concentrating the probability in small intervals just to the left of each knot point (to make the function right continuous). Given:

```
dsev [xs] [ps]
```

where `xs` and `ps` are the vectors of outcomes and probabilities, internally `aggregate` creates:

```
xss = np.sort(np.hstack((xs - 2 ** -30, xs)))
pss = np.vstack((ps1, np.zeros_like(ps1))).reshape((-1,), order='F')[:-1]
fz_discr = ss.rv_histogram((pss, xss))
```

The value `2**-30` needs to be smaller than the bucket size resolution, i.e., enough not to "split the bucket". The mass is to the left of the knot to make a right continuous function (the approximation ramps up before the knot). Generally histograms are downsampled, not upsampled, so this is not a restriction.

A `dsev` statement is translated into the more general:

```
sev dhistorgram xps [xs] [ps]
```

where `dhistrogram` creates a discrete histogram (as above) and the `xps` keyword prefixes inputting the knots and probabilities. It is also possible to specify the input severity as a continuous histogram that is uniform on $(x_k, x_{k+1}]$. The discrete probabilities are $p_k = P(x_k < X \le x_{k+1})$. To create a rv_histogram variable is much easier, just use:

```
sev chistorgram xps [xs] [ps]
```

which is translated into:

```
xs2 = np.hstack((xs, xs[-1] + xs[1]))
fz_cts = ss.rv_histogram((ps2, xs2))
```

The code adds an additional knot at the end to create enough differences (there are only two differences between three points). The num robertson uses a `chistogram`.

The discrete method is appropriate when the distribution will be used and interpreted as fully discrete, which is the assumption the FFT method makes and the default. The continuous method is useful if the distribution will be used to create a scipy.stats rv_histogram variable. If the continuous method is interpreted as discrete and if the mean is computed as $\sum_i p_k x_k$, which is appropriate for a discrete variable, then it will be under-estimated by $b/2$.

Parametric Severity

A parametric distribution can be specified in two ways:

```
sev DIST_NAME MEAN cv CV
sev DIST_NAME <SHAPE1> <SHAPE2>
```

where

- `sev` is a keyword indicating the severity specification,
- `DIST_NAME` is the `scipy.stats` distribution name, see *scipy.stats Continuous Random Variables*,
- `MEAN` is the expected loss,
- `cv` (lowercase) is a keyword indicating entry of the CV,
- `CV` is the loss coefficient of variation, and
- `SHAPE1`, `SHAPE2` are the (optional) shape variables.

The first form enters the expected ground-up severity and CV directly. It is available for distributions with only one shape parameter and the beta distribution on $[0, 1]$. `aggregate` uses a formula (lognormal, gamma, beta) or numerical method (all other one shape parameter distributions) to solve for the shape parameter to achieve the correct CV and then scales to the desired mean. The second form directly enters the shape variable(s). Shape parameters entered for zero parameter distributions are ignored.

Example. Entering `sev lognorm 10 cv 0.2` produces a lognormal distribution with a mean of 10 and a CV of 0.2. Entering `lognorm 0.2` produces a lognormal with $\mu = 0$ and $\sigma = 0.2$, which can then be *scaled and shifted*.

`DIST_NAME` can be any zero, one, or two shape parameter `scipy.stats` continuous distribution. They have (mostly) easy to guess names. For example:

- Distributions with no shape parameters include: `norm`, Gaussian normal; `unif`, uniform; and `expon`, the exponential.
- Distributions with one shape parameter include: `pareto`, `lognorm`, `gamma`, `invgamma`, `loggamma`, and `weibull_min` the Weibull.
- Distributions with two shape parameters include: `beta` and `gengamma`, the generalized gamma.

See *scipy.stats Continuous Random Variables* for a full list and list of distributions for details of each.

Details.

`dhistogram` and `chistogram` create discrete (point mass) and continuous (ogive) empirical distributions. `chistogram` is rarely used and `dhistogram` is easier to input using `dsev`, *Non-Parametric Severity Distributions*.

Shifting and Scaling Severity

A parametric severity clause can be transformed by scaling and location factors, following the `scipy.stats` `scale` and `loc` syntax. Location is a shift or translation. The syntax is:

```
sev SCALE * DISTNAME SHAPE + LOC
sev SCALE * DISTNAME SHAPE - LOC
```

For zero parameter distributions `SHAPE` is omitted. Two parameter distributions are entered `sev SCALE * DISTNAME SHAPE1 SHAPE2 + LOC`.

Examples.

- `sev lognorm 10 cv 3`: lognormal, mean 10, CV 0.
- `sev 10 * lognorm 1.75`: lognormal, $10X$, $X \sim \text{lognormal}(\mu = 0, \sigma = 1.75)$
- `sev 10 * lognorm 1.75 + 20`: lognormal, $10X + 20$
- `sev 10 * lognorm 1 cv 3 + 50`: lognormal: $10Y + 50$, $Y \sim$ lognormal mean 1, CV 3
- `sev 100 * pareto 1.3 - 100`: Pareto, shape $\alpha = 3$, scale $\lambda = 100$.
- `sev 100 * pareto 1.3`: Single parameter Pareto for $x \geq 100$, Shape (α) 3, scale (λ) 100
- `sev 50 * norm + 100`: normal, mean (location) 100, standard deviation (scale) 50. No shape parameter.
- `sev 5 * expon`: exponential, mean (scale) 5. No shape parameter.
- `sev 5 * uniform + 1`: uniform between 1 and 6, scale 5, location 1. No shape parameters.
- `sev 50 * beta 2 3`: beta: $50Z$, $Z \sim \beta(2, 3)$, shape parameters 2, 3, scale 50.

With this parameterization, the Pareto has survival function $S(x) = (100/(100 + x))^{1.3}$.

The scale and location parameters can be *vectors*.

Warning: `dsev` severities **cannot** be shifted or scaled. If that is required use a Python f-string to adjust the outcomes:

```
f'dsev [{{5 * outcomes + 10}}] [probabilities]'
```

Warning: Shifting left (negative shift) must be written with space `sev 10 * lognorm 1.5 - 10` not `sev 10 * lognorm 1.5 -10`. The lexer binds uniary minus to the number, so the latter omits the operator. `sev 10 * lognorm 1.5 + -10`, `sev 10 * lognorm 1.5 +10` and `sev 10 * lognorm 1.5 + 10` are all acceptable because there is no unary +. This is a known bug and is insidious: the `-10` will be interpreted as a second shape parameter and ignored. You will not get the answer you expect.

Unconditional Severity

The severity clause is entered ground-up. It is converted to a distribution conditional on a loss to the layer if there is a limits sub-clause. Thus, for an excess layer y xs a, the severity used to create the aggregate has a distribution $X \mid X > a$, where X is specified in the `sev` clause. For a ground-up (or missing) layer there is no adjustment.

The default behavior can be over-ridden by adding `!` after the severity distribution.

Example.

The default behavior uses severity conditional to the layer. In this example, the conditional layer severity is 6.

```
In [1]: from aggregate import build, qd

In [2]: cond = build('agg DecL:Conditional '
   ...:              '1 claim '
   ...:              '12 xs 8 '
   ...:              'sev 20 * uniform '
   ...:              'fixed')
   ...:

In [3]: qd(cond)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)      Skew(X) Est Skew(X)
X
Freq     1                            0
Sev      6        6 -9.992e-16 0.57735   0.57735 -8.2046e-15 -9.0251e-14
Agg      6        6 -9.992e-16 0.57735   0.57735 -9.5721e-15 -9.0251e-14
log2 = 16, bandwidth = 1/2048, validation: fails sev skew, agg skew.
```

To specify unconditional severity, append `!` to the severity clause. The unconditional layer severity is only 3.6 because there is just a 60% chance of attaching the layer. In the last line, `uncd.sevs[0].fz` is `sev 20 * uniform` ground-up.

```
In [4]: uncd = build('agg DecL:Unconditional '
   ...:              '1 claim '
   ...:              '12 xs 8 '
   ...:              'sev 20 * uniform ! '
   ...:              'fixed')
   ...:

In [5]: qd(uncd)

      E[X] Est E[X]     Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     1                            0
Sev    3.6      3.6 -1.1102e-16 1.1055    1.1055 0.65784     0.65784
Agg    3.6      3.6 -1.1102e-16 1.1055    1.1055 0.65784     0.65784
log2 = 16, bandwidth = 1/2048, validation: not unreasonable.

In [6]: print(uncd.sevs[0].fz.sf(8), uncd.agg_m / cond.agg_m)
0.6 0.6
```

`scipy.stats` Continuous Random Variables

All `scipy.stats` continuous random variable classes can be used as severity distributions, see *scipy.stats Severity Distributions* for a complete list. As always, with great power comes great responsibility.

Warning: The user must determine if a severity distribution is appropriate, `aggregate` will not check! Only specified zero parameter (uniform, exponential, normal) and two parameter () distributions are allowed, but **all** one parameter distributions will work. However, any zero parameter distribution can be called with a dummy argument, that is ignored. **Be careful out there!**

2.4.5 The Frequency Clause

The exposure and severity clauses determine the expected claim count. The `frequency` clause specifies the other particulars of the claim count distribution. As with severity, the syntax is different for non-parametric and parametric distributions.

Non-Parametric Frequency Distributions

An exposure clause:

```
dfreq [outcomes] <[probabilities]>
```

directly specifies the frequency distribution. The `outcomes` and `probabilities` are specified as in *Non-Parametric Severity Distributions*. There is no need for a frequency clause at the end.

Examples.

```
agg A dfreq [1 2 3] [.5 3/8 1/8] sev lognorm 50 cv 1.75
agg A dfreq [1 2 3] [.5 3/8 1/8] dsev [1:11]
```

The first specifies a frequency distribution with outcomes 1, 2, or 3 occurring with probabilities 0.5, 0.375, and 0.125 respectively. Probabilities can be entered as decimals or fractions. The second combines a non-parametric frequency and severity.

Parametric Frequency Distributions

The following parametric frequency distributions are supported. Remember that the exposure clause determines the expected claim count.

- `poisson`, no additional parameters required.
- `geometric`, no additional parameters required.
- `fixed`, no additional parameters required, expected claim count must be an integer.
- `bernoulli`, no additional parameters required; expected claim count must be ≤ 1.
- `binomial SHAPE`, the shape parameter sets p and $n = \mathsf{E}[N]/p$.
- `neyman SHAPE` (or `neymana` or `neymanA`), the Neyman A Poisson-compound Poisson. The shape variable gives the average number of claimants per claim. See JKK and Consul and Shenton [1973].
- `pascal SHAPE1 SHAPE2` (the generalized Poisson-Pascal, see REF), where `SHAPE1` gives the cv and `SHAPE2` the number of claims per occurrence.

Example.

```
agg A 100 claims sev lognorm 50 cv 0.75 poisson
agg A 100 claims sev lognorm 50 cv 0.75 mixed gamma 0.2
```

specifies a Poisson frequency. and negative binomial frequency respectively. For the latter, frequency CV equals `(1 + .2**2 * 100) ** .5 / 10 = 0.22361`.

Mixed-Poisson Frequency Distributions

A G-mixed Poisson frequency (see *Mixed Frequency Distributions*), where G has expectation 1, can be specified using the `mixed` keyword, followed by the name and shape parameters of the mixing distribution:

```
mixed DIST_NAME SHAPE1 <SHAPE2>
```

`SHAPE1` specifies cv of the mixing distribution. The following mixing distributions are supported:

- `gamma SHAPE1` is a gamma-Poisson, i.e., negative binomial. Since the mix mean (shape times scale) equals one $\alpha\beta = 1$ and hence the mix variance equals $c := \alpha = (cv)^{-2}$, which is sometimes called the contagion. The negative binomial variance equals $n(1 + cn)$.
- `delaporte SHAPE1 SHAPE2`, a shifted gamma and the second parameter equals the proportion of certain claims (which determines a minimum claim count).
- `ig SHAPE1` the inverse Gaussian distribution
- `sig SHAPE1 SHAPE2` the shifted inverse Gaussian, parameter 2 as for Delaporte.
- `beta SHAPE1` a beta-Poisson with mean 1 and cv `SHAPE1`. Use with caution.
- `sichel SHAPE1 SHAPE2` is Sichel's (generalized inverse Gaussian) distribution with `SHAPE2` equal to λ.
  - `sichel.gamma SHAPE1` is the same as Delaporte
  - `sichel.ig SHAPE1` is the same as a shifted inverse Gaussian.

Example.

```
agg A 100 claims sev lognorm 50 cv 0.75 mixed gamma 0.2
```

specifies a negative binomial (gamma-mixed Poisson) frequency respectively. The variance equals $100 \times (1 + 0.2^2 \times 100)$ and the CV equals `(1 + .2**2 * 100) ** .5 / 10 = 0.22361`.

Warning: Fixed frequency will accept non-integer input, but will not return a distribution (it will have negative probabilities). Be careful!

Zero Modification and Zero Truncation

Todo: Not yet implemented.

2.4.6 Mixed Severity Distributions

Prerequisites: Examples use `build` and `qd`, and basic `Aggregate` output.

The variables in the severity clause (scale, location, distribution ID, shape parameters, mean and CV) can be vectors to create a **mixed severity** distribution. All elements are broadcast against one-another.

Example:

```
sev lognorm 1000 cv [0.75 1.0 1.25 1.5 2] wts [0.4, 0.2, 0.1, 0.1, 0.1]
```

expresses a mixture of five lognormals, each with a mean of 1000 and CVs equal to 0.75, 1.0, 1.25, 1.5, and 2, and with weights 0.4, 0.2, 0.1, 0.1, 0.1. Equal weights are expressed as `wts=5`, or the relevant number of components (note equals sign). A missing weights clause is interpreted as giving each severity weight 1 which results in five times the total loss. Commas in the lists are optional.

Warning: Weights are applied to exposure, and their meaning depends on how exposure is entered.

If exposure is given by claim count, then the weights apply to claim count. This gives the usual mixture of severity curves. However, if exposure is entered as loss or premium times a loss ratio, then the weights give the proportion of expected loss, not the claim count. **Make sure the weights are appropriate to the way exposure is expressed**. For example, if the mixture is used to split small and large claims, then an 80/20 split small/large claim counts may well correspond to a 20/80 split of expected losses (Pareto rule of thumb).

Example.

This example illustrates the different behaviors of `wts`. The weights adjust claim counts for each mixture component when exposures are given by claims.

```
In [1]: from aggregate import build, qd

In [2]: a01 = build('agg DecL:01 '
   ...:             '100 claims '
   ...:             '5000 xs 0 '
   ...:             'sev lognorm [10 20 50 75 100] '
   ...:             'cv [0.75 1.0 1.25 1.5 2] '
   ...:             'wts [0.4, 0.25, 0.15, 0.1, 0.1] '
   ...:             'poisson'
   ...:             , bs=1/2, approximation='exact')
   ...:

In [3]: qd(a01)

       E[X] Est E[X]   Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    100                         0.1                 0.1
Sev   33.978    33.978 2.5414e-09  2.3807    2.3807   15.161      15.161
Agg   3397.8    3397.8 2.5414e-09 0.25822   0.25822   1.2927      1.2927
log2 = 16, bandwidth = 1/2, validation: not unreasonable.
```

Mixed severity with Poisson frequency is the same as the sum of five independent components. The `report_df` shows the mixture details.

```
In [4]: qd(a01.report_df.iloc[:, :-3])

view               0        1        2        3        4  independent
statistic
name         DecL:01  DecL:01  DecL:01  DecL:01  DecL:01      DecL:01
limit           5000     5000     5000     5000     5000         5000
attachment         0        0        0        0        0            0
el               400      500      750   749.93   997.86       3397.8
freq_m            40       25       15       10       10          100
freq_cv      0.15811      0.2   0.2582  0.31623  0.31623          0.1
freq_skew    0.15811      0.2   0.2582  0.31623  0.31623          0.1
sev_m             10       20       50   74.993   99.786       33.978
sev_cv          0.75        1   1.2498   1.4942   1.9175       2.3807
sev_skew      2.6719        4   5.6619   7.1636   8.3826       15.161
agg_m            400      500      750   749.93   997.86       3397.8
agg_cv       0.19764  0.28284  0.41329  0.56857  0.68386      0.25822
agg_skew     0.30882  0.56568    1.054   1.7191    2.224       1.2927
```

This aggregate can also be built as a `Portfolio`.

```
In [5]: a02 = build(
   ...:     'port DecL:02 '
   ...:         'agg Unit1 40 loss 5000 xs 0 sev lognorm 10 cv 0.75 poisson '
   ...:         'agg Unit2 25 loss 5000 xs 0 sev lognorm 20 cv 1.00 poisson '
```

(continues on next page)

(continued from previous page)

```
   ...:         'agg Unit3 15 loss 5000 xs 0 sev lognorm 50 cv 1.25 poisson '
   ...:         'agg Unit4 10 loss 5000 xs 0 sev lognorm 75 cv 1.50 poisson '
   ...:         'agg Unit5 10 loss 5000 xs 0 sev lognorm 100 cv 2.00 poisson '
   ...:     , bs=1/2, approximation='exact')
   ...:

In [6]: qd(a02)

              E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
unit  X
Unit1 Freq       4                           0.5                  0.5
      Sev       10        10  8.9604e-09    0.75    0.75014    2.6719       2.6704
      Agg       40        40  8.9606e-09   0.625    0.62504   0.97656      0.97653
Unit2 Freq    1.25                       0.89443              0.89443
      Sev       20        20 -9.7705e-09       1          1         4       3.9997
      Agg       25        25 -9.7705e-09  1.2649     1.2649    2.5298       2.5298
Unit3 Freq     0.3                        1.8257               1.8257
      Sev       50        50 -1.0074e-09  1.2498     1.2498    5.6619       5.6618
      Agg       15        15 -1.0074e-09  2.9224     2.9224    7.4526       7.4526
Unit4 Freq 0.13335                        2.7385               2.7385
      Sev   74.993    74.993 -4.0382e-10  1.4942     1.4942    7.1636       7.1635
      Agg       10        10 -4.0383e-10  4.9237     4.9237    14.887       14.887
Unit5 Freq 0.10021                        3.1589               3.1589
      Sev   99.786    99.786  1.1018e-08  1.9175     1.9175    8.3826       8.3826
      Agg       10        10  1.1019e-08  6.8313     6.8313    22.216       22.216
total Freq  5.7836                       0.41582              0.41582
      Sev    17.29     17.29  2.0519e-09  2.2699                   26
      Agg      100       100   2.031e-09  1.0314     1.0314    8.7339       8.7338
log2 = 16, bandwidth = 1/2, validation: not unreasonable.
```

Actual frequency equals total frequency times weight. Setting `wts=5` results in equal weights, here 0.2.

```
In [7]: a03 = build('agg DecL:03 '
   ...:             '100 claims '
   ...:             '5000 xs 0 '
   ...:             'sev lognorm [10 20 50 75 100] '
   ...:             'cv [0.75 1.0 1.25 1.5 2] '
   ...:             ' wts=5 '
   ...:             'poisson'
   ...:             , bs=1/2, approximation='exact')
   ...:

In [8]: qd(a03)

       E[X]  Est E[X]   Err E[X]   CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq    100                          0.1                  0.1
Sev  50.956    50.956 3.5836e-09  2.1341     2.1341    11.891       11.891
Agg  5095.6    5095.6 3.5835e-09 0.23568    0.23568   0.99494      0.99494
log2 = 16, bandwidth = 1/2, validation: not unreasonable.
```

Missing weights are set to 1, resulting in five times loss. This behavior is generally not what you want!

```
In [9]: a04 = build('agg DecL:04 '
   ...:             '100 claims '
   ...:             '5000 xs 0 '
   ...:             'sev lognorm [10 20 50 75 100] '
   ...:             'cv [0.75 1.0 1.25 1.5 2] '
   ...:             'poisson'
   ...:             , bs=1, approximation='exact')
   ...:
```

(continues on next page)

(continued from previous page)

```
In [10]: qd(a04)

       E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    500                      0.044721           0.044721
Sev  50.956   50.956 -2.2323e-08   2.1341    2.1341   11.891      11.891
Agg   25478    25478 -2.2323e-08   0.1054    0.1054  0.44495     0.44495
log2 = 16, bandwidth = 1, validation: not unreasonable.
```

If exposures are determined via losses (directly or using premium and loss ratio or exposure and rate), then the weights apply to expected loss. The resulting mixture is quite different.

```
In [11]: a01e = build('agg DecL:01e '
   ....:              f'{a01.agg_m} loss '
   ....:              '5000 xs 0 '
   ....:              'sev lognorm [10 20 50 60 70] '
   ....:              'cv [0.75 1.0 1.25 1.5 2] '
   ....:              'wts [0.4, 0.25, 0.15, 0.1, 0.1] '
   ....:              'poisson'
   ....:              , bs=1/2, approximation='exact')
   ....:

In [12]: qd(a01e)

       E[X] Est E[X]   Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  115.2                     0.093168           0.093168
Sev  29.493   29.493 1.1694e-08   2.1101    2.1101   14.661      14.661
Agg  3397.8   3397.8 1.1694e-08  0.21755   0.21755    1.113       1.113
log2 = 16, bandwidth = 1/2, validation: not unreasonable.

In [13]: qd(a01e.report_df.iloc[:, :-3])

view               0         1         2         3         4 independent
statistic
name         DecL:01e  DecL:01e  DecL:01e  DecL:01e  DecL:01e    DecL:01e
limit            5000      5000      5000      5000      5000        5000
attachment          0         0         0         0         0           0
el             460.82    576.02    864.03     691.2    805.71      3397.8
freq_m         46.082    28.801    17.281     11.52     11.52       115.2
freq_cv       0.14731   0.18634   0.24056   0.29462   0.29462    0.093168
freq_skew     0.14731   0.18634   0.24056   0.29462   0.29462    0.093168
sev_m              10        20        50    59.997    69.938      29.493
sev_cv           0.75         1    1.2498    1.4968    1.9523      2.1101
sev_skew       2.6719         4    5.6619    7.3823    9.6027      14.661
agg_m          460.82    576.02    864.03     691.2    805.71      3397.8
agg_cv        0.18414   0.26352   0.38505   0.53034   0.64624     0.21755
agg_skew      0.28772   0.52703   0.98195    1.6404    2.3418       1.113
```

Mixed Exponential Distributions

The mixed exponential distribution (MED) is used by major US rating bureaus to model severity and compute increased limits factors (ILFs). This example explains how to create a MED in `aggregate`. The distribution is initially created as an `Aggregate` object with a degenerate frequency identically equal to 1 claim to focus on the severity. We then explain how frequency mixing interacts with a mixed severity.

The next table of exponential means and weights appears on slide 24 of Li Zhu, Introduction to Increased Limits Factors, 2011 RPM Basic Ratemaking Workshop,, titled a "Sample of Actual Fitted Distribution". At the time, it was a reasonable curve for US commercial auto. We will use these means and weights.

Mean	Weight
2,763	0.824796
24,548	0.159065
275,654	0.014444
1,917,469	0.001624
10,000,000	0.000071

Here the DecL to create this mixture.

```
In [14]: med = build('agg Decl:MED '
   ....:             '1 claim '
   ....:             'sev [2.764e3 24.548e3 275.654e3 1.917469e6 10e6] * '
   ....:             'expon 1 '
   ....:             'wts [0.824796 0.159065 0.014444 0.001624, 0.000071] '
   ....:             'fixed')
   ....:

In [15]: qd(med)

      E[X] Est E[X]  Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     1                         0
Sev  13990    13798 -0.013708 12.034    12.203  103.79      103.76
Agg  13990    13798 -0.013708 12.034    12.203  103.79      103.76
log2 = 16, bandwidth = 4000, validation: fails sev mean, agg mean.
```

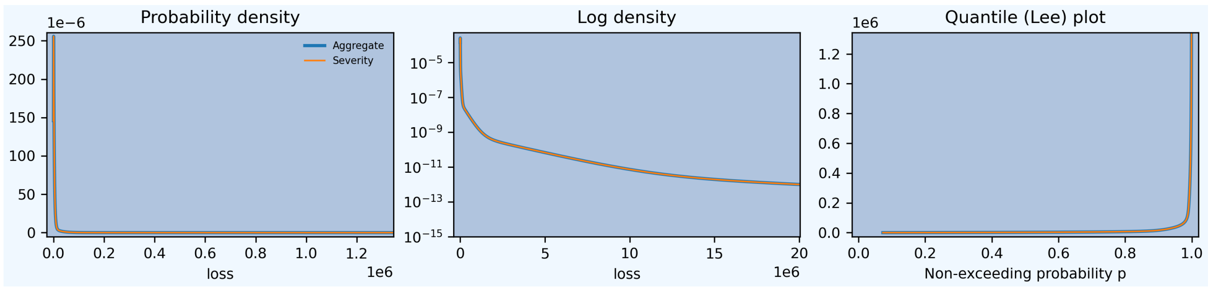
Note: Currently, it is necessary to enter a dummy shape parameter 1 for the exponential, even though it does not take a shape. This is a known bug in the parser.

The exponential distribution is surprisingly thick-tailed. It can be regarded as the dividing line between thin and thick tailed distributions. In order to achieve good accuracy, the modeling increases the number of buckets to 2^{18} (i.e., `log2=18`) and uses a bucket size `bs=500`. The dataframe `report_df` is a more detailed version of the audit dataframe that includes information from `statistics_df` about each severity component. (The reported claim counts are equal to the weights and cannot be interpreted as fixed frequencies. They can be regarded as frequencies for a Poisson or mixed Poisson.)

```
In [16]: med.update(log2=18, bs=500)

In [17]: qd(med)

      E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     1                           0
Sev  13990    13987 -0.00022819 12.034    12.037  103.79      103.72
Agg  13990    13987 -0.00022819 12.034    12.037  103.79      103.72
log2 = 18, bandwidth = 500, validation: fails sev mean, agg mean.
```

The middle diagnostic plot, the log density, shows the mixture components.

```
In [18]: med.plot()
```

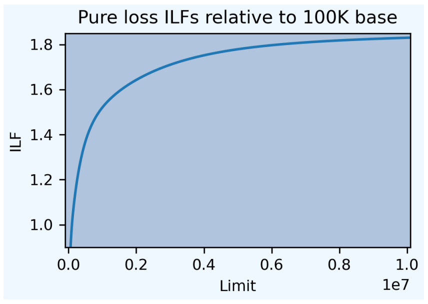


The `density_df` dataframe includes a column `lev`. From this we can pull out ILFs. Zhu reports the ILF at 1M equals 1.52.

```
In [19]: qd(med.density_df.loc[1000000, 'lev'] / med.density_df.loc[100000, 'lev'])
1.5204
```

Here is a graph of the ILFs by limit.

```
In [20]: base = med.density_df.loc[100000, 'lev']

In [21]: ax = (med.density_df.lev / base).plot(xlim=[-100000,10.1e6], ylim=[0.9, 1.
↪85],
   ....:                                         figsize=(3.5, 2.45))
   ....:

In [22]: ax.set(xlabel='Limit', ylabel='ILF', title='Pure loss ILFs relative to␣
↪100K base');
```



Saving to the Knowledge

We can save the MED severity in the knowledge and then refer to it by name.

```
In [23]: build('sev COMMAUTO [2.764e3 24.548e3 275.654e3 1.917469e6 10e6] * '
   ....:       ' expon 1 wts [0.824796 0.159065 0.014444 0.001624, 0.000071]');
   ....:

In [24]: a05 = build('agg DecL:05 [20 8 4 2] claims [1e6, 2e6 5e6 10e6] xs 0 '
   ....:             'sev sev.COMMAUTO fixed',
   ....:             log2=18, bs=500)
   ....:

In [25]: qd(a05)

          E[X]   Est E[X]     Err E[X]  CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
X
Freq        34                              0
Sev      11973      11970 -0.00026506  6.328     6.3298   31.665       31.665
```

(continues on next page)

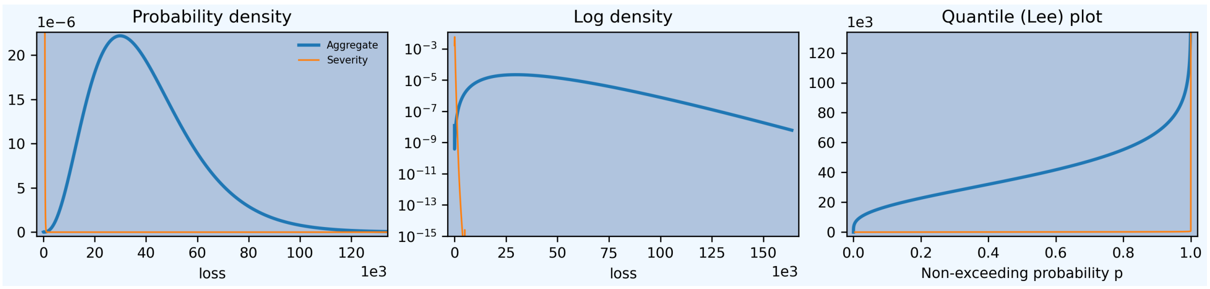
(continued from previous page)

```
Agg  4.0708e+05 4.0697e+05 -0.00026506 1.0852    1.0855  5.4306      5.4305
log2 = 18, bandwidth = 500, validation: fails sev mean, agg mean.
```

Different Distributions

The kind of distribution can vary across mixtures. In the following, exposure varies for each curve, rather than using weights, see *Vectorization: Limit Profiles and Mixed Severity*.

```
In [26]: a06 = build('agg DecL:06 [100 200] claims '
   ....:             '5000 xs 0 '
   ....:             'sev [gamma lognorm] [100 150] cv [1 0.5] '
   ....:             'mixed gamma 0.5',
   ....:             log2=16, bs=2.5)
   ....:

In [27]: qd(a06.report_df.iloc[:, :-2])

view                0        1 independent    mixed
statistic
name          DecL:06  DecL:06     DecL:06  DecL:06
limit            5000     5000        5000     5000
attachment          0        0           0        0
el              10000    30000       40000    40000
freq_m            100      200         300      300
freq_cv        0.5099  0.50498     0.37712  0.50332
freq_skew      1.0002        1     0.80295        1
sev_m             100      150      133.33   133.33
sev_cv              1      0.5     0.65551  0.65551
sev_skew            2    1.625      1.4508   1.4508
agg_m           10000    30000       40000    40000
agg_cv        0.51962  0.50621     0.40127  0.50474
agg_skew       1.0022   1.0002     0.88112   1.0001

In [28]: a06.plot()
```



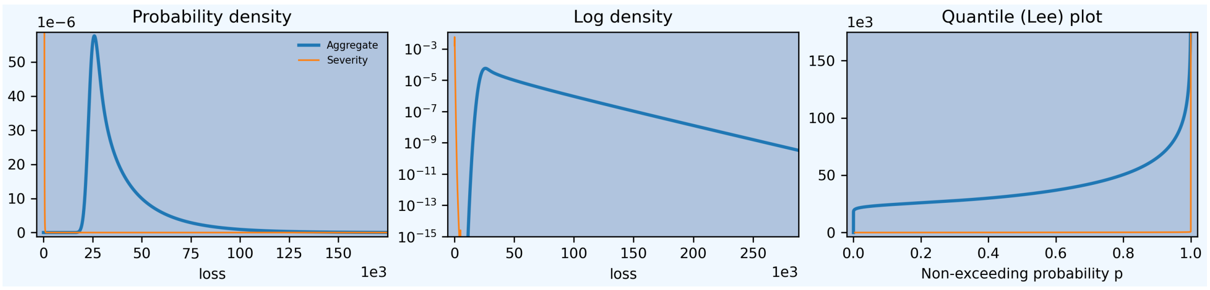
Using a Delaporte (shifted) gamma mixing often produces more realistic output than a gamma, avoiding very good years.

```
In [29]: a07 = build('agg DecL:07 [100 200] claims '
   ....:              '5000 xs 0 '
   ....:              'sev [gamma lognorm] [100 150] cv [1 0.5] '
   ....:              'mixed delaporte 0.5 0.6',
   ....:             log2=18, bs=2.5)
   ....:

In [30]: qd(a07.report_df.iloc[:, :-2])

view                0        1 independent    mixed
statistic
```

(continues on next page)

(continued from previous page)

```
name         DecL:07  DecL:07    DecL:07  DecL:07
limit           5000     5000       5000     5000
attachment         0        0          0        0
el             10000    30000      40000    40000
freq_m           100      200        300      300
freq_cv       0.5099  0.50498    0.37712  0.50332
freq_skew     2.4145   2.4561     1.9682   2.4705
sev_m            100      150     133.33   133.33
sev_cv             1      0.5    0.65551  0.65551
sev_skew           2    1.625     1.4508   1.4508
agg_m          10000    30000      40000    40000
agg_cv       0.51962  0.50621    0.40127  0.50474
agg_skew      2.3386   2.4456     2.1507   2.4582

In [31]: a07.plot()
```



Severity Mixtures and Mixed Frequency

All severity components in an aggregate share the same frequency mixing value, inducing correlation between the parts. An Aon study, Aon Benfield [2015], shows that commercial auto has parameter uncertainty CV around 25%. Building with

```
In [32]: a08 = build('agg DecL:08 '
   ....:             '500 claims '
   ....:             '500000 xs 0 sev sev.COMMAUTO '
   ....:             'poisson'
   ....:             , approximation='exact')
   ....:

In [33]: a09 = build('agg DecL:09 '
   ....:             '500 claims '
   ....:             '500000 xs 0 sev sev.COMMAUTO '
   ....:             'mixed gamma 0.25'
   ....:             , approximation='exact')
   ....:

In [34]: qd(a08)

           E[X]    Est E[X]     Err E[X]     CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq        500                           0.044721            0.044721
Sev       10266       10266 -4.9496e-05    3.9519     3.9522    9.4914       9.4913
Agg  5.1332e+06 5.1329e+06 -4.9496e-05   0.18231    0.18232   0.41833      0.41833
log2 = 16, bandwidth = 200, validation: not unreasonable.

In [35]: qd(a09)

           E[X]    Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
```

(continues on next page)

(continued from previous page)

```
Freq        500                         0.25397            0.50006
Sev       10266      10264 -0.0001979  3.9519     3.9528   9.4914        9.491
Agg  5.1332e+06 5.1321e+06 -0.0001979 0.30941    0.30943  0.55969       0.5597
log2 = 16, bandwidth = 400, validation: fails sev mean, agg mean.
```

The effect of shared mixing is shown in `report_df`. In order to focus on the mixing and ease the computational burden, apply a 500,000 policy limit to model a self-insured retention. Assume a claim count of 500 claims; for smaller portfolios the impact of mixing is less pronounced because idiosyncratic process risk dominates.

The `independent` column in `report_df` shows statitics assuming the mixture components are independent; `mixed` includes the effect of shared mixing variables.

The next block shows results with a Poisson frequency, where there is no mixing. The independent and mixed columns are identical.

```
In [36]: qd(a08.report_df.drop(['name']).iloc[:, :-2])

view                 0          1          2          3          4  independent     ␣
↪mixed
statistic                                                                          ␣
↪
limit            5e+05      5e+05      5e+05      5e+05      5e+05        5e+05     ␣
↪5e+05
attachment           0          0          0          0          0            0     ␣
↪     0
el          1.1399e+06 1.9524e+06 1.6662e+06 3.5738e+05      17314   5.1332e+06 5.
↪1332e+06
freq_m           412.4     79.532      7.222      0.812     0.0355          500     ␣
↪   500
freq_cv       0.049243    0.11213    0.37211     1.1097     5.3074     0.044721   0.
↪044721
freq_skew     0.049243    0.11213    0.37211     1.1097     5.3074     0.044721   0.
↪044721
sev_m             2764      24548 2.3072e+05 4.4013e+05 4.8771e+05        10266     ␣
↪10266
sev_cv               1          1    0.73847    0.29449    0.12909       3.9519    ␣
↪3.9519
sev_skew             2          2    0.39392     -2.071    -5.6054       9.4914    ␣
↪9.4914
agg_m       1.1399e+06 1.9524e+06 1.6662e+06 3.5738e+05      17314   5.1332e+06 5.
↪1332e+06
agg_cv         0.06964    0.15858    0.46257     1.1569     5.3515      0.18231    0.
↪18231
agg_skew       0.10446    0.23787    0.54133     1.1826     5.3739      0.41833    0.
↪41833
```

This block shows mixed gamma (negative binomial) frequency. There are two differences: the individual components have higher CVs (they asymptotically approach 25% for a large portfolio), and the mixed column includes correlation between units (aggregate CV is greater than independent). Glenn Meyers had the idea of using shared mixing variables to ensure aggregate portfolio dynamics are not influenced by how the portfolio is split into units.

```
In [37]: qd(a09.report_df.drop(['name']).iloc[:, :-2])

view                 0          1          2          3          4  independent     ␣
↪mixed
statistic                                                                          ␣
↪
limit            5e+05      5e+05      5e+05      5e+05      5e+05        5e+05     ␣
↪5e+05
attachment           0          0          0          0          0            0     ␣
↪     0
```

(continues on next page)

(continued from previous page)

```
el          1.1399e+06 1.9524e+06 1.6662e+06 3.5738e+05      17314  5.1332e+06 5.
↪1332e+06
freq_m           412.4     79.532      7.222      0.812     0.0355         500    ␣
↪  500
freq_cv         0.2548      0.274    0.44829     1.1376     5.3133     0.21474    0.
↪25397
freq_skew      0.50009     0.5021    0.58771     1.1925     5.3251       0.473    0.
↪50006
sev_m             2764      24548 2.3072e+05 4.4013e+05 4.8771e+05       10266    ␣
↪10266
sev_cv               1          1    0.73847    0.29449    0.12909      3.9519    ␣
↪3.9519
sev_skew             2          2    0.39392     -2.071    -5.6054      9.4914    ␣
↪9.4914
agg_m       1.1399e+06 1.9524e+06 1.6662e+06 3.5738e+05      17314  5.1332e+06 5.
↪1332e+06
agg_cv         0.25952    0.29605    0.52581     1.1836     5.3573     0.22858    0.
↪30941
agg_skew       0.50102    0.51935     0.6983     1.2604     5.3913     0.42255    0.
↪55969
```

2.4.7 Limit Profiles

Prerequisites: Examples use `build` and `qd`, and basic `Aggregate` output.

All exposure variables can be vectors. This feature makes it easy to express a **limit profile**. All exposure related elements (claim count, premium, loss, loss ratio) are broadcast against one-another.

Example:

```
agg Eg1                                \
[1000 2000 4000 1000] premium at 0.65 lr \
[1000 2000 5000 4000] xs [0 0 0 1000]    \
sev lognorm 500 cv 1.25                \
mixed gamma 0.6
```

expresses a limit profile with 1000 of premium at 1000 xs 0; 2000 at 2000 xs 0, 4000 at 5000 xs 0, and 1000 at 4000 xs 1000. In this case all the loss ratios are the same, but they could vary too.

A (mixed) compound Poisson aggregate with a mixed severity is a sum of aggregates, with the mixture weights applied to the expected claim count. This is analogous to the fact that $\exp(a + b) = \exp(a)\exp(b)$. In terms of a compound Poisson,

$$\mathsf{CP}(\lambda, \sum w_i F_i) =_d \sum_i \mathsf{CP}(w_i\lambda, F_i)$$

where $=_d$ indcates the two sides have the same distribution. For Poisson frequency, the components on the right are independent; for mixed frequencies they are not.

In this case, we have selected a mixed frequency, using a gamma CV 0.6 mixing distribution. All of the limits share the same mixing variable. The effect of this is shown in the `report_df`, comparing the independent and mixed columns. The former adds the mixture components independently whereas the latter uses the common mixing variable. The increase in aggregate CV is quite marked.

```
In [1]: from aggregate import build, qd

In [2]: a10 = build('agg DecL:10 '
   ...:             '[1000 2000 4000 1000] premium at 0.65 lr '
   ...:             '[1000 2000 5000 4000] xs [0 0 0 1000] '
   ...:             'sev lognorm 500 cv 1.25 '
```

(continues on next page)

(continued from previous page)

```
   ...:              'mixed gamma 0.6')
   ...:

In [3]: qd(a10)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 10.473                      0.67489             1.2083
Sev  496.51   496.51 -8.8063e-09  1.1061    1.1061   3.1179      3.1179
Agg    5200     5200 -8.9414e-09 0.75651   0.75651   1.3155      1.3155
log2 = 16, bandwidth = 1, validation: not unreasonable.

In [4]: qd(a10.report_df.iloc[:, :-2])

view              0        1        2        3 independent    mixed
statistic
name        DecL:10  DecL:10  DecL:10  DecL:10     DecL:10  DecL:10
limit          1000     2000     5000     4000      3518.1   3518.1
attachment        0        0        0     1000      83.724   83.724
el              650     1300     2600      650        5200     5200
freq_m       1.5833     2.77    5.243  0.87685      10.473   10.473
freq_cv     0.99579  0.84913  0.74211   1.2249     0.47078  0.67489
freq_skew    1.3573   1.2731   1.2272   1.5188     0.80136   1.2083
sev_m        410.54   469.32    495.9   741.29      496.51   496.51
sev_cv      0.74534  0.96384   1.1639   1.1443      1.1061   1.1061
sev_skew    0.77863   1.8007   3.4165    2.019      3.1179   3.1179
agg_m           650     1300     2600      650        5200     5200
agg_cv       1.1587   1.0278  0.89949   1.7302      0.5797  0.75651
agg_skew     1.6154   1.5795   1.5449   2.5962       1.019   1.3155
```

2.4.8 Vectorization: Limit Profiles and Mixed Severity

Prerequisites: Examples use `build` and `qd`, and basic `Aggregate` output.

Using a Limit Profile with a Mixed Severity

Limit Profiles and *Mixed Severity Distributions* can be combined. Each mixed severity is applied to each limit profile component.

sub-components.

Example.

This example combines three limit bands and a severity with two mixture components. It creates an aggregate with six severities. The `report_df` dataframe shows the components (transposed extract shown). The mixture weights apply to claim counts, since exposure is specified by number of expected claims.

```
In [1]: from aggregate import build, qd

In [2]: a11 = build('agg DecL:11 '
   ...:              '[10 20 30] claims '
   ...:              '[100 200 75] xs [0 50 75] '
   ...:              'sev lognorm 100 cv [1 2] wts [0.6 0.4] '
   ...:              'poisson')
   ...:

In [3]: qd(a11)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
```

(continues on next page)

(continued from previous page)

```
X
Freq    60                       0.1291             0.1291
Sev   60.65     60.65 -8.3258e-08 0.76525    0.76525    1.2624       1.2624
Agg    3639      3639 -8.3262e-08 0.16256    0.16256   0.21483      0.21483
log2 = 16, bandwidth = 1/8, validation: not unreasonable.

In [4]: qd(a11.report_df.loc[['limit', 'attachment', 'freq_m',
   ...:    'agg_m', 'agg_cv']].T.iloc[:-4])
   ...:

statistic limit attachment freq_m  agg_m  agg_cv
view
0           100          0      6 406.32 0.44721
1           100          0      4 210.35  0.6055
2           200         50 13.618 984.54 0.35857
3           200         50 6.3822 552.26 0.51379
4            75         75  20.23 969.19 0.25552
5            75         75 9.7703 516.34 0.35838
```

Example.

We can combine the mixed exponential from *Mixed Exponential Distributions* with a limits profile.

```
In [5]: from aggregate import build, qd

In [6]: a12 = build('agg DecL:12 [20 8 4 2] claims [1e6, 2e6 5e6 10e6] xs 0 '
   ...:                  'sev [2.764e3 24.548e3 275.654e3 1.917469e6 10e6] * '
   ...:                  'expon 1 wts [0.824796 0.159065 0.014444 0.001624, 0.
↪000071] fixed',
   ...:                  log2=18, bs=500)
   ...:

In [7]: qd(a12)

           E[X]    Est E[X]     Err E[X]   CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
X
Freq          34                                0
Sev        11973       11970 -0.00026506   6.328     6.3298   31.665       31.665
Agg   4.0708e+05 4.0697e+05 -0.00026506 1.0852     1.0855   5.4306       5.4305
log2 = 18, bandwidth = 500, validation: fails sev mean, agg mean.
```

The `report_df` shows all 20 components: 4 limits x 5 mixture components.

```
In [8]: qd(a12.report_df.loc[['limit', 'attachment', 'freq_m',
   ...:    'agg_m', 'agg_cv']].T.iloc[:-4])
   ...:

statistic limit attachment   freq_m  agg_m  agg_cv
view
0         1e+06          0   16.496  45595 0.24621
1         1e+06          0   3.1813  78095 0.56066
2         1e+06          0  0.28888  77515  1.7165
3         1e+06          0  0.03248  25309  2.3031
4         1e+06          0  0.00142 1351.3  4.8442
5         2e+06          0   6.5984  18238  0.3893
6         2e+06          0   1.2725  31238 0.88648
7         2e+06          0  0.11555  31830  2.9287
8         2e+06          0 0.012992  16133   5.082
9         2e+06          0 0.000568 1029.6  10.827
10        5e+06          0   3.2992 9118.9 0.55055
11        5e+06          0  0.63626  15619  1.2537
```

(continues on next page)

(continued from previous page)

```
12        5e+06          0 0.057776  15926  4.1603
13        5e+06          0 0.006496  11538  10.463
14        5e+06          0 0.000284 1117.5  24.125
15        1e+07          0   1.6496 4559.5  0.7786
16        1e+07          0  0.31813 7809.5   1.773
17        1e+07          0 0.028888 7963.1  5.8836
18        1e+07          0 0.003248 6194.1  17.135
19        1e+07          0 0.000142 897.61  47.664
```

Circumventing Products: Modeling Multiple Units in One Aggregate

When severity weights sum to one, the severity is treated as a mixture and all exposure terms are broadcast against all severity terms in an outer product.

When severity weights are missing or sum to the number of severity components (e.g., are all equal to 1) the result is an item by item combination, circumventing the outer product. There are two cases when this alternative is useful:

1. Two or more units each with a different severity, but with a shared mixing variable. For example, to model two units with expected losses 100 and 200, one with a gamma mean 10 CV 1 severity and the other lognormal mean 15 CV 1.5 and both share a gamma mixing variable:

   ```
   agg MixedPremReserve                     \
   [100 200] claims                         \
   sev [gamma lognorm] [10 15] cv [1 1.5]   \
   mixed gamma 0.4
   ```

   The result should be the two-way combination, not the four-way exposure and severity product.

2. Exposures with different limits may have different severity curves. Again, the limit profile and severity curves should all be broadcast together at once, rather than broadcasting limits and severities separately and then taking the outer product:

   ```
   agg Eg4                                       \
   [10 10 10] claims                             \
   [1000 2000 5000] xs 0                         \
   sev lognorm [50 100 150] cv [0.1 0.15 0.2]    \
   poisson
   ```

Example.

The next two examples illustrate the different behavior.

1. Two units with different limits and severities and no weights. `report_df` shows only two components modeled.

```
In [9]: a13 = build('agg DecL:13 '
   ...:             '[10 20] claims '
   ...:             '[1000 2000] xs 0 '
   ...:             'sev [gamma lognorm] [10 15] cv [1 1.5] '
   ...:             'mixed gamma 0.4 ')
   ...:

In [10]: qd(a13)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     30                       0.4397            0.80358
Sev  13.333   13.333 -4.0697e-07  1.4543    1.4543    8.255       8.255
Agg     400      400 -4.0707e-07 0.51365   0.51365    1.014       1.014
log2 = 16, bandwidth = 1/16, validation: not unreasonable.
```

(continues on next page)

(continued from previous page)

```
In [11]: qd(a13.report_df.loc[['limit', 'attachment', 'freq_m',
   ....:     'agg_m', 'agg_cv']].T.iloc[:-4])
   ....:

statistic limit attachment freq_m agg_m  agg_cv
view
0          1000          0     10   100     0.6
1          2000          0     20   300 0.56778
```

1. Adding weights results in a mixed severity, 80% for the gamma and 20% for lognormal. Now `report_df` shows that each limit band is combined with each severity, resulting in four modeled components.

```
In [12]: a14 = build('agg DecL:14 '
   ....:             '[10 20] claims '
   ....:             '[1000 2000] xs 0 '
   ....:             'sev [gamma lognorm] [10 15] cv [1 1.5] '
   ....:             'wts [.8 .2] '
   ....:             'mixed gamma 0.4 ')
   ....:

In [13]: qd(a14)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    30                       0.4397            0.80358
Sev     11       11 -2.9593e-07  1.2362    1.2362   7.7928      7.7928
Agg    330      330 -2.1305e-06 0.49424   0.49422  0.94603     0.94464
log2 = 16, bandwidth = 1/32, validation: not unreasonable.

In [14]: qd(a14.report_df.loc[['limit', 'attachment', 'freq_m',
   ....:     'agg_m', 'agg_cv']].T.iloc[:-4])
   ....:

statistic limit attachment freq_m  agg_m  agg_cv
view
0          1000          0      8     80 0.64031
1          1000          0      2 29.997  1.3328
2          2000          0     16    160 0.53385
3          2000          0      4     60 0.98584
```

2.4.9 The Reinsurance Clauses

Prerequisites: Excess of loss reinsurance terminology.

Occurrence and aggregate reinsurance can be specified in a way similar to limits and deductibles. Both clauses are optional. The ceded or net position can be output. Layers can be stacked and can include co-participations.

The options for both clauses are:

- Keywords `ceded to` or `net of` determine which losses flow out of the reinsurance.
- A fraction `po` limit `xs` attachment describes a partial placement, e.g., `0.5 so 3 xs 2`.
- A participation `so` limit `xs` attachment describes a partial placement by the ceded limit, e.g., `1 po 3 xs 2`. This syntax is equivalent to `0.333 so 3 xs 2`.

An unlimited cover is denoted `inf`. Shares of unlimited covers must be expressed as shares, for obvious reasons.

Layers can be stacked using the `and` keyword. The initial `net of` or `ceded to` applies to all layers in the tower.

The occurrence reinsurance clause comes after severity but before frequency, because you need to know severity but not frequency. The aggregate clause comes after frequency. If frequency is specified using `dfreq` the occurrence clause comes before the aggregate clause.

The syntax is best illustrated with some examples.

Examples.

1. **Occurrence reinsurance**:

   ```
   agg Trucking                        \
   5000 loss 1000 xs 0                 \
   sev lognorm 50 cv 1.75              \
   occurrence net of 750 xs 250        \
   poisson
   ```

   specifies the distribution of losses to the net position on the Trucking policy after a per occurrence cession of the 750 xs 250 layer. This net position can also be written using limits and attachments rather than reinsurance:

   ```
   agg Trucking                        \
   ?? loss                             \
   250 xs 0                            \
   sev lognorm 50 1.75                 \
   poisson
   ```

   for some level of losses. Running:

   ```
   agg Trucking                        \
   5000 loss 1000 xs 0                 \
   sev lognorm 50 cv 1.75              \
   occurrence ceded to 750 xs 250      \
   poisson
   ```

   models ceded losses.

2. **Aggregate reinsurance**:

   ```
   agg WorkComp                        \
   15000 loss                          \
   500 xs 0                            \
   sev lognorm 50 cv 1.75              \
   poisson                             \
   aggregate ceded to 50% so 2000 xs 15000
   ```

   specifies the distribution of losses ceded to an aggregate protection for the 2000 xs 15000 layer of total losses (attaching at the loss pick), with occurrences limited to 500. The underlying business could be an SIR on a large account Workers Compensation policy, and the aggregate is a part of the insurance charge (Table L, M).

3. **Occurrence and aggregate reinsurance**:

   ```
   agg Trucking 5000                   \
   loss 1000 xs 0                      \
   sev lognorm 50 cv 1.75              \
   occurrence net of 50% so 250 xs 250 and 500 xs 500   \
   poisson                             \
   aggregate net of 250 po 1000 xs 4000 and 5000 xs 5000
   ```

   applies two occurrence and two aggregate layers to the Trucking portfolio. The 250 xs 250 occurrence layer is only 50% placed (`so` stands for share of), and the second is 100% (by default) of 500 xs 500. The net of the occurrence programs flows through to aggregate layers, 250 part of 1000 xs 4000 (25% placement, `po` stands for part of), and 100% share of the 5000 xs 5000 aggregate layers. The modeled outcome is net of all four layers. In this case, it is not possible to write the net of occurrence using limits and attachments.

Note: All occurrence reinsurance assumes free and unlimited reinstatements.

Layering Losses in a Tower

Underwriters are often interested in layering out losses from ground-up to the policy limit. For example, a 5M limit may be layered as 250 xs 0, 250 xs 250, 500 xs 500, 1000 xs 1000, and 3000 xs 2000. A tower can be input manually:

```
occurrence ceded to 250 xs 0 and 250 xs 250 and 500 xs 500  \
and 1000 xs 1000 and 3000 xs 2000
```

There is also a shorthand for layering, since it is quite common. A layering can be entered by specifying just the layer break points using the `tower` keyword:

```
occurrence ceded to tower [0 250 500 1000 2000 5000]
```

The tower does not have to start at 0 and does not have to exhaust the entire policy limit. Towers can be applied to occurrence and aggregate reinsurance.

See *reinsurance pricing* for more examples, including an approach to reinstatements.

2.4.10 The Note Clause

An optional note (comment) on the distribution can be added at the end, as the last clause. The note can include hints for computation. The text is enclosed in braces.

```
note{US Prems Ops, light hazard severity; for ABC account; recommend:- log2:16, bs:
↪1/32}
```

Notes cannot include a line break.

2.4.11 The `tweedie` Keyword

Prerequisites: Tweedie distribution from GLMs. Use of `build`.

See also: ../../5_technical_guides/5_x_tweedie.

The `aggregate` language keyword `tweedie` makes it easy to build Tweedie distributions. It uses reproductive parameters μ, p, σ^2 (mean, power, and dispersion), since these are most natural for GLM modeling.

Example.

The keyword is used as follows to produce $\mathsf{Tw}_{1.05}(2,5)$, mean 2, $p = 1.05$, and dispersion 5.

```
In [1]: from aggregate import build, qd, mv

In [2]: a15 = build('agg DecL:15 tweedie 2 1.05 5')

In [3]: qd(a15)

        E[X] Est E[X]     Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 0.40671                         1.568              1.568
Sev   4.9175   4.9175 -2.3315e-15 0.22942   0.22942  0.45883     0.45883
Agg        2        2 -2.2204e-15  1.6088    1.6088   1.6892      1.6892
log2 = 16, bandwidth = 1/512, validation: not unreasonable.

In [4]: mv(a15)
mean      = 2
```

(continues on next page)

(continued from previous page)

```
variance = 10.35265
std dev  = 3.21755

In [5]: print(f'Expected variance = disp x mean ** p = {2**1.05 * 5:.5f}')
Expected variance = disp x mean ** p = 10.35265
```

Inspecting the (non-trivial parts of the) specification shows the parser converts it into the additive form:

```
In [6]: {k: v for k,v in a15.spec.items()
   ...:     if v!=0 and v is not None and v!=''}
   ...:
Out[6]:
{'name': 'DecL:15',
 'exp_en': 0.4067100332315139,
 'exp_limit': inf,
 'sev_name': 'gamma',
 'sev_a': 18.999999999999982,
 'sev_scale': 0.2588162309603446,
 'sev_wt': 1,
 'sev_ub': inf,
 'sev_conditional': True,
 'freq_name': 'poisson',
 'freq_p0': nan,
 'note': 'Tw(p=1.05, μ=2.0, σ^2=5.0) --> CP(λ= 0.40671, ga(α=19, β=0.25881623),␣
↪scale=0.25881623'}
```

The note shows the compound Poisson specification.

The helper function `tweedie_convert` translates between parameterizations. The scale (dispersion) parameter σ^2 has offsetting effects: higher σ^2 results in a lower claim count, a higher gamma mean, and a more skewed aggregate distribution with a bigger mass at zero.

Example.

The code below shows the three Tweedie representations, starting with the easiest to interpret.

```
In [7]: from aggregate import tweedie_convert

In [8]: import pandas as pd

In [9]: p = 1.005; μ = 1; σ2 = 0.1;                       \
   ...: m0 = tweedie_convert(p=p, μ=μ, σ2=σ2);            \
   ...: λ = μ**(2-p) / ((2-p) * σ2);                      \
   ...: α = (2 - p) / (p - 1);                            \
   ...: β = μ / (λ * α);                                  \
   ...: tw_cv = σ2**.5 * μ**(p/2-1);                      \
   ...: sev_m = α *  β;                                   \
   ...: sev_cv = α**-0.5;                                 \
   ...: m1 = tweedie_convert(λ=λ, m=sev_m, cv=sev_cv); \
   ...: m2 = tweedie_convert(λ=λ, α=α, β=β);
   ...:

In [10]: assert np.allclose(m0, m1, m2)

In [11]: temp = pd.concat((m0, m1, m2), axis=1); \
   ....: temp.columns = ['mean p disp', 'lambda sev m cv', 'lambda shape scale'];
   ....:

In [12]: with pd.option_context('display.float_format', lambda x: f'{x:.12g}'):
   ....:     print(temp)
   ....:
             mean p disp   lambda sev m cv  lambda shape scale
```

(continues on next page)

(continued from previous page)

```
μ                       1                 1                   1
p                   1.005             1.005               1.005
σ^2                   0.1               0.1                 0.1
λ           10.0502512563     10.0502512563       10.0502512563
α                     199               199                 199
β                  0.0005            0.0005              0.0005
tw_cv      0.316227766017    0.316227766017      0.316227766017
sev_m              0.0995            0.0995              0.0995
sev_cv    0.0708881205008   0.0708881205008     0.0708881205008
p0      4.31748997327e-05 4.31748997327e-05   4.31748997327e-05
```

Three different ways of specifying the same Tweedie distribution.

```
In [13]: program = f'''
   ....: agg DecL:16 {λ} claims sev gamma {sev_m:.8g} cv {sev_cv} poisson
   ....: agg DecL:17 {λ} claims sev {β:.4g} * gamma {α:.4g} poisson
   ....: agg DecL:18 tweedie {μ} {p} {σ2}
   ....: '''
   ....:

In [14]: tweedies = build(program)

In [15]: for a in tweedies:
   ....:     print(a.program)
   ....:     qd(a.object.describe)
   ....:     print()
   ....:
agg DecL:16 10.050251256281404 claims sev gamma 0.0995 cv 0.07088812050083283␣
↪poisson

       E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)  Err CV(X)  Skew(X)  Est␣
↪Skew(X)
X                                                                              ␣
↪
Freq  10.05       NaN         NaN  0.31544        NaN        NaN  0.31544      ␣
↪NaN
Sev   0.0995    0.0995 -8.2379e-14 0.070888   0.070888   3.12e-06  0.14178      0.
↪14177
Agg       1         1 -1.1036e-13  0.31623    0.31623 1.5599e-08  0.31781      0.
↪31781

agg DecL:17 10.050251256281404 claims sev 0.0005 * gamma 199 poisson

       E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)  Err CV(X)  Skew(X)  Est␣
↪Skew(X)
X                                                                              ␣
↪
Freq  10.05       NaN         NaN  0.31544        NaN        NaN  0.31544      ␣
↪NaN
Sev   0.0995    0.0995 -1.0825e-13 0.070888   0.070888   3.12e-06  0.14178      0.
↪14177
Agg       1         1 -1.3756e-13  0.31623    0.31623 1.5599e-08  0.31781      0.
↪31781

agg DecL:18 tweedie 1 1.005 0.1

       E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)  Err CV(X)  Skew(X)  Est␣
↪Skew(X)
X                                                                              ␣
↪
Freq  10.05       NaN         NaN  0.31544        NaN        NaN  0.31544      ␣
```

(continues on next page)

(continued from previous page)

```
↪NaN
Sev   0.0995     0.0995 -8.6597e-14  0.070888    0.070888    3.12e-06   0.14178       0.
↪14177
Agg        1          1   -1.148e-13   0.31623     0.31623  1.5599e-08   0.31781       0.
↪31781
```

Convert from reproductive form:

```
In [16]: tweedie_convert(p=1.05, μ=2, σ2=5)
Out[16]:
μ           2.000000
p           1.050000
σ^2         5.000000
λ           0.406710
α          19.000000
β           0.258816
tw_cv       1.608777
sev_m       4.917508
sev_cv      0.229416
p0          0.665837
dtype: float64
```

Convert from additive form:

```
In [17]: tweedie_convert(λ=0.406710033, m=4.917508388, cv=0.229415734)
Out[17]:
μ           2.000000
p           1.050000
σ^2         5.000000
λ           0.406710
α          19.000000
β           0.258816
tw_cv       1.608777
sev_m       4.917508
sev_cv      0.229416
p0          0.665837
dtype: float64
```

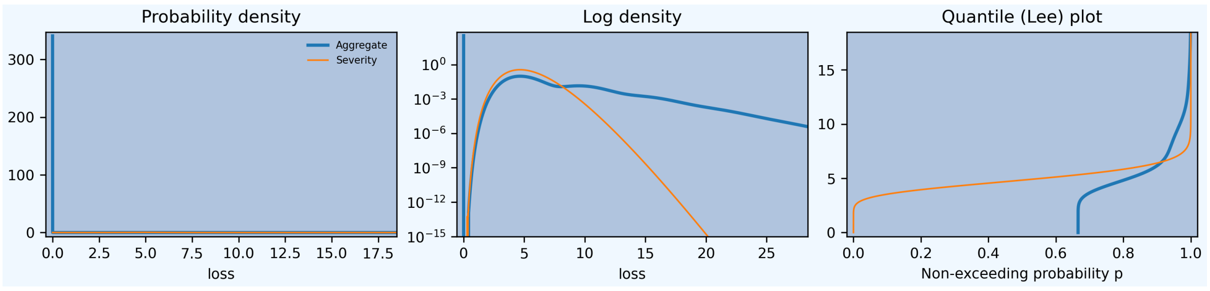
Build a Tweedie using reproductive parameters, `p`, `mu`, `sigma2`.

```
In [18]: a19 = build('agg DecL:19 tweedie 2 1.05 5')

In [19]: a19.plot()

In [20]: qd(a19)

        E[X]  Est E[X]     Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq 0.40671                           1.568               1.568
Sev    4.9175    4.9175 -2.3315e-15  0.22942    0.22942  0.45883      0.45883
Agg         2          2 -2.2204e-15   1.6088     1.6088   1.6892       1.6892
log2 = 16, bandwidth = 1/512, validation: not unreasonable.

In [21]: print(a19.spec)
{'name': 'DecL:19', 'exp_el': 0, 'exp_premium': 0, 'exp_lr': 0, 'exp_en': 0.
↪4067100332315139, 'exp_attachment': None, 'exp_limit': inf, 'sev_name': 'gamma',
↪'sev_a': 18.999999999999982, 'sev_b': 0, 'sev_mean': 0, 'sev_cv': 0, 'sev_loc':␣
↪0, 'sev_scale': 0.2588162309603446, 'sev_xs': None, 'sev_ps': None, 'sev_wt': 1,
↪'sev_lb': 0, 'sev_ub': inf, 'sev_conditional': True, 'sev_pick_attachments':␣
↪None, 'sev_pick_losses': None, 'occ_reins': None, 'occ_kind': '', 'freq_name':
↪'poisson', 'freq_a': 0, 'freq_b': 0, 'freq_zm': False, 'freq_p0': nan, 'agg_reins
```

(continues on next page)

(continued from previous page)

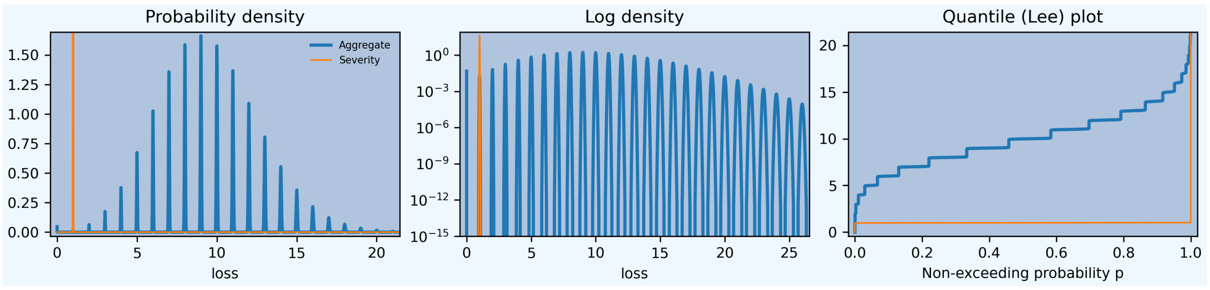
```
↪': None, 'agg_kind': '', 'note': 'Tw(p=1.05, μ=2.0, σ^2=5.0) --> CP(λ= 0.40671,␣
↪ga(α=19, β=0.25881623), scale=0.25881623'}

In [22]: print(a19.cdf(0), np.exp(-.40671))
0.6658372329829731 0.6658372551097528
```



Example.

When p is close to 1, the Tweedie approaches a Poisson. Here mean = 10 and sigma2 = 1, so the distribution is not over-dispersed. The gamma severity has mean 1 and a very small CV; it acts like degenerate distribution at 1.

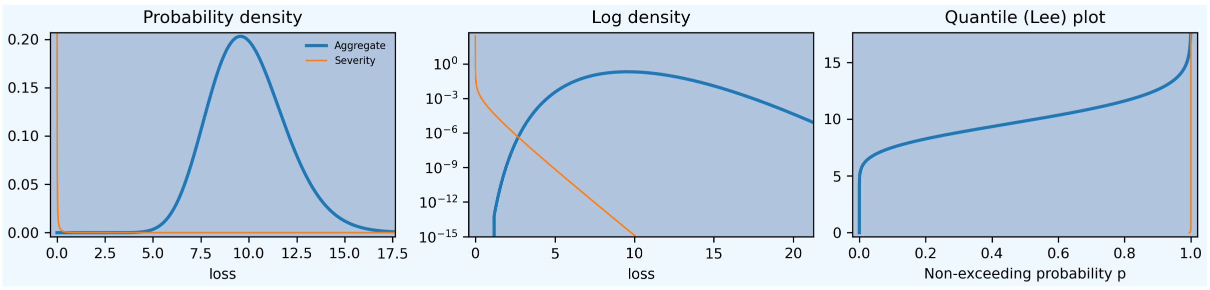
```
In [23]: a20 = build('agg DecL:20 tweedie 10 1.0001 1')

In [24]: a20.plot()

In [25]: qd(a20)

       E[X] Est E[X]   Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 9.9987                        0.31625            0.31625
Sev  1.0001    1.0001 1.9973e-11 0.010001  0.010004 0.019989    0.019977
Agg      10        10  1.997e-11  0.31626   0.31626   0.3163      0.3163
log2 = 16, bandwidth = 1/1024, validation: not unreasonable.

In [26]: tweedie_convert(p=1.0001, μ=10, σ2=1)
Out[26]:
μ            10.000000
p             1.000100
σ^2           1.000000
λ             9.998698
α          9999.000000
β             0.000100
tw_cv         0.316264
sev_m         1.000130
sev_cv        0.010001
p0            0.000045
dtype: float64
```



Example.

When p is close to 2, the Tweedie approaches a Gamma. Here mean = 10, and sigma2=0.04. The variance equals `sigma2 mu^2`, so CV = sigma = 0.2

```
In [27]: a21 = build('agg DecL:21 tweedie 10 1.999 0.04', log2=16, bs=1/256)

In [28]: a21.plot()

In [29]: qd(a21)

          E[X]    Est E[X]    Err E[X]      CV(X)  Est CV(X)    Skew(X)  Est Skew(X)
X
Freq       25058                           0.0063173            0.0063173
Sev   0.00039908 0.00039773 -0.0033706     31.607     31.715     63.214        63.21
Agg           10      9.9663 -0.0033706    0.19977    0.20045    0.39934      0.39932
log2 = 16, bandwidth = 1/256, validation: fails sev mean, agg mean.
```



Build the same distribution explicitly from gamma severities. Here the gamma is built using mean and CV or shape and scale.

```
In [30]: tc = tweedie_convert(p=1.9999, μ=10, σ2=.04)

In [31]: print(tc)
μ              10.000000
p               1.999900
σ^2             0.040000
λ          250057.571255
α               0.000100
β               0.399868
tw_cv           0.199977
sev_m           0.000040
sev_cv         99.995000
p0              0.000000
dtype: float64

In [32]: m, cv = tc['μ'], tc['tw_cv']

In [33]: print(m, cv)
10.0 0.19997697547449372

In [34]: g = build(f'sev g gamma {m} cv {cv}')

In [35]: sh = cv ** -2; sc = m / sh

In [36]: print(sc, sh)
0.39990790719926267 25.005757125520617

In [37]: g2 = build(f'sev g2 {sc} * gamma {sh}')

In [38]: print(g2.stats(), g.stats())
(9.999999999999968, 3.999079071994487) (10.000000000000108, 3.999079071991673)
```

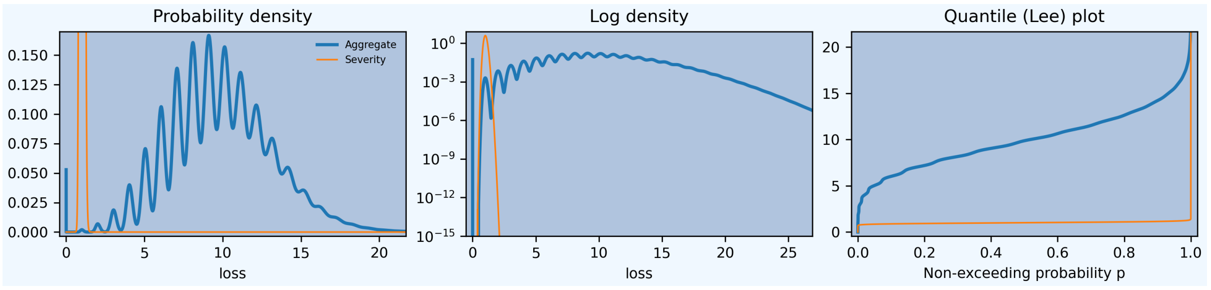
Analytic Error Analysis

There is a series expansion for the pdf of a Tweedie computed by conditioning on the number of claims and using that a convolution of gammas with the same scale parameter is again gamma. For a Tweedie with expected frequency λ, gamma shape α and scale β, it is given by

$$f(x) = \sum_{n\ge 1} e^{-\lambda}\frac{\lambda^n}{n!}\frac{x^{n\alpha-1}e^{-x/\beta}}{\Gamma(n\alpha)\beta^{n\alpha}}$$

for $x > 0$ and $f(x) = \exp(-\lambda)$. The exact function shows the FFT method is very accurate.

```
In [39]: from aggregate import tweedie_convert, build, qd

In [40]: from scipy.special import loggamma

In [41]: import matplotlib.pyplot as plt

In [42]: import numpy as np

In [43]: from pandas import option_context

In [44]: a = build('agg Tw tweedie 10 1.01 1')

In [45]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 9.8711                      0.31829           0.31829
Sev  1.0131   1.0131 -1.9318e-14  0.1005    0.1005  0.20101     0.20101
Agg      10       10 -2.2427e-14 0.31989   0.31989  0.32309     0.32309
log2 = 16, bandwidth = 1/1024, validation: not unreasonable.

In [46]: a.plot()
```



A Tweedie with p close to 1 approximates a Poisson. Its gamma severity is very peaked around its mean (high α and offsetting small β).

The next function provides a transparent, if inefficient, implementation of the Tweedie density.

```
In [47]: def tweedie_density(x, mean, p, disp):
   ....:     pars = tweedie_convert(p=p, μ=mean, σ2=disp)
   ....:     λ = pars['λ']
   ....:     α = pars['α']
   ....:     β = pars['β']
   ....:     if x == 0:
   ....:         return np.exp(-λ)
   ....:     logl = np.log(λ)
   ....:     logx = np.log(x)
   ....:     logb = np.log(β)
   ....:     logbase = -λ
   ....:     log_term = 100
   ....:     const = -λ - x / β
```

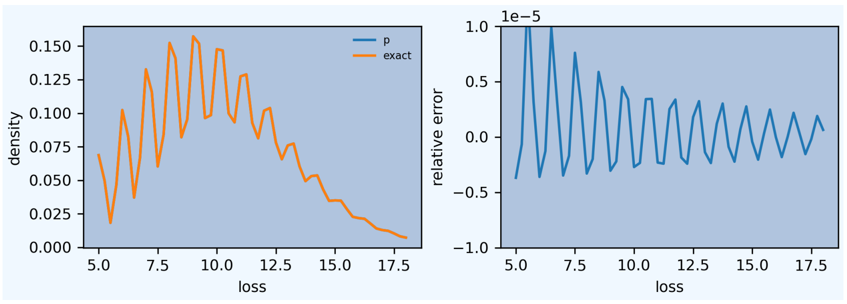
(continues on next page)

(continued from previous page)

```
   ....:     ans = 0.0
   ....:     for n in range(1, 2000): #while log_term > -20:
   ....:         log_term = (const  +
   ....:                     + n * logl  +
   ....:                     + (n * α - 1) * logx +
   ....:                     - loggamma(n+1) +
   ....:                     - loggamma(n * α) +
   ....:                     - n * α * logb)
   ....:         ans += np.exp(log_term)
   ....:         if n > 20 and log_term < -227:
   ....:             break
   ....:     return ans
   ....:
```

The following graphs show that the FFT approximation is excellent, across a wide range, just as its good moment-matching performance suggests it would be.

```
In [48]: bit = a.density_df.loc[5:a.q(0.99):256, ['p']]

In [49]: bit['exact'] = [tweedie_density(i, 10, 1.01, 1) for i in bit.index]

In [50]: bit['p'] /= a.bs

In [51]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True, squeeze=True)

In [52]: ax0, ax1 = axs.flat

In [53]: bit.plot(ax=ax0);

In [54]: ax0.set(ylabel='density');

In [55]: bit['err'] = bit.p / bit.exact - 1

In [56]: bit.err.plot(ax=ax1);

In [57]: ax1.set(ylabel='relative error', ylim=[-1e-5, 1e-5]);
```



2.4.12 Summary of Objects Created by DecL

Objects created by `build()` in the DecL guide.

```
In [1]: from aggregate import build, pprint_ex

In [2]: for n, r in build.qlist('^DecL:').iterrows():
   ...:     pprint_ex(r.program, split=20)
   ...:
```

2.5 Individual Risk Pricing

Objectives: Applications of the `Aggregate` class to individual risk pricing, including LEVs, ILFs, layering, and the insurance charge and savings (Table L, M), illustrated using problems from CAS Part 8.

Audience: Individual risk large account pricing, broker, or risk retention actuary.

Prerequisites: DecL, underwriting and insurance terminology, aggregate distributions, risk measures.

See also: *Reinsurance Pricing*, *The Reinsurance Clauses*. For other related examples see *Published Problems and Examples*, especially *Bahnemann Monograph*. *Self-Insurance Plan Stop-Loss Insurance* for theoretical background.

Contents:

The examples in this section are illustrative. `aggregate` gives the gross, ceded, and net distributions and with those in hand, it is possible to answer any reasonable question about a large account program.

2.5.1 Helpful References

- Fisher *et al.* [2019]
- Bahnemann [2015]
- Other CAS Part 8 readings.

2.5.2 Insurance Charge and Insurance Savings in `Aggregate`

Creating a custom table of insurance charges and savings, varying with account size, specific occurrence limit, and entry ratio (aggregate limit) is very easy using `aggregate`. We will make a custom function to illustrate one solution.

First, we need a severity curve. This step is very important, and would be customized to the state and hazard group distribution of expected losses. We use a simple mixture of a lognormal for small claims and a Pareto for large claims, with a mean of about 25 (work in 000s). Create it as an object in the knowledge using `build()`. The parameters are selected judgmentally.

```
In [1]: from aggregate import build, qd

In [2]: mu, sigma, shape, scale, wt = \
   ...:     -0.204573975,  1.409431871, 1.633490596, 57.96737143, 0.742942461
   ...:

In [3]: mean = wt * np.exp(mu + sigma**2 / 2) + (1 - wt) * scale / (shape - 1)

In [4]: build(f'sev IR:WC '
   ...:       f'[exp({mu}) {scale}] * [lognorm pareto] [{sigma} {shape}] '
   ...:       f'+ [0 {-scale}] wts [{wt} {1-wt}]');
   ...:

In [5]: print(f'Mean = {mean:.1f} in 000s')
Mean = 25.2 in 000s
```

Second, we will build the model for a large account with 350 expected claims and an occurrence limit of 100M. This model is used to set the update parameters. Assume a gamma mixed Poisson frequency distribution with a mixing CV of 25% throughout. The CV could be an input parameter in a production application.

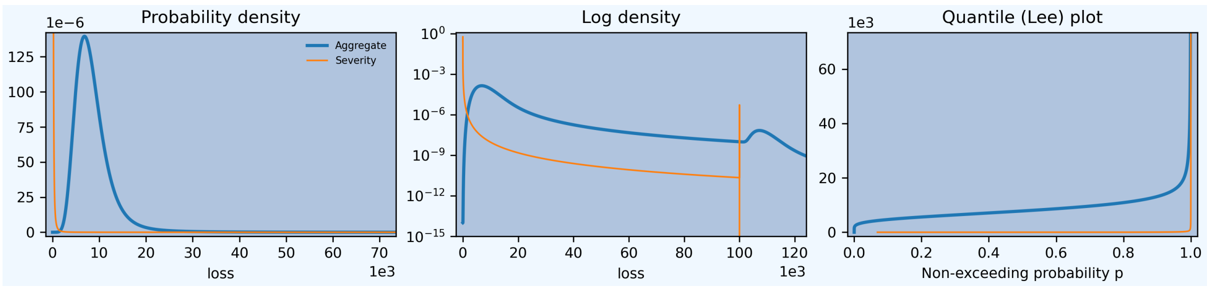
```
In [6]: a01 = build('agg IR:Base '
   ...:             '350 claims '
   ...:             '100000 xs 0 '
   ...:             'sev sev.IR:WC '
   ...:             'mixed gamma 0.25 ',
   ...:             update=False)
   ...:

In [7]: qd(a01)

       E[X]   CV(X)  Skew(X)
X
Freq    350 0.25565  0.50012
Sev  24.947  10.172   177.37
Agg  8731.6  0.6008   7.3319
log2 = 0, bandwidth = na, validation: n/a, not updated.

In [8]: qd(a01.statistics.loc['sev', [0, 1, 'mixed']])

name          0          1      mixed
measure
ex1      2.2005      90.69     24.947
ex2      35.299 2.5281e+05      65013
ex3      4127.8 1.1293e+10 2.9029e+09
mean     2.2005      90.69     24.947
cv        2.508     5.4532     10.172
skew     23.299     92.803     177.37
```

Look at the `aggregate_error_analysis` to pick `bs` (see *Estimating and Testing bs For Aggregate Objects*). Use an expanded number of buckets `log2=19` because the mixture includes small mean lognormal and large mean Pareto components (some trial and error not shown).

```
In [9]: err_anal = a01.aggregate_error_analysis(19)

In [10]: qd(err_anal, sparsify=False)

view      agg    est     abs          rel         rel         rel
stat        m      m       m            m           h       total
bs
0.0625 8731.6 8552.1 -179.54    -0.020562 0.0012526   -0.021815
0.1250 8731.6 8647.1 -84.529   -0.0096808 0.0025053   -0.012186
0.2500 8731.6 8731.1 -0.5447 -6.2382e-05 0.0050105 -0.0050729
0.5000 8731.6 8728.8  -2.798 -0.00032045  0.010021   -0.010342
1.0000 8731.6 8719.9 -11.715  -0.0013417  0.020042   -0.021384
2.0000 8731.6 8694.4 -37.172  -0.0042572  0.040084   -0.044341
4.0000 8731.6 8639.2 -92.465    -0.01059  0.080168   -0.090758
```

Select `bs=1/4` as the most accurate from the displayed range (`` ('rel', 'm')``). Update and plot. The plot shows the impact of the occurrence limit in the extreme right tail.

```
In [11]: a01.update(approximation='exact', log2=19, bs=1/4, normalize=False)

In [12]: qd(a01)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    350                      0.25565            0.50012
Sev  24.947   24.946 -5.5033e-05  10.172    10.172   177.37      177.37
Agg  8731.6   8731.1 -6.2382e-05  0.6008   0.60063   7.3319      7.3121
log2 = 19, bandwidth = 1/4, validation: not unreasonable.

In [13]: a01.plot()
```



Third, create a custom function of account size and the occurrence limit, to produce the `Aggregate` object and a small table of insurance savings and charges. Account size is measured by the expected ground-up claim count. It should be clear how to extend this function to include custom severity, different mixing CVs, or produce factors for different entry ratios. The answer is returned in a `namedtuple`.

```
In [14]: from collections import namedtuple

In [15]: def make_table(claims=360, occ_limit=100000):
   ....:     """
   ....:     Make a table of insurance charges and savings by entry ratio for
   ....:     specified account size (expected claim count) and specific
   ....:     occurrence limit.
   ....:     """
   ....:     a01 = build(f'agg IR:{claims}:{occ_limit} '
   ....:                 f'{claims} claims '
   ....:                 f'{occ_limit} xs 0 '
   ....:                  'sev sev.IR:WC '
   ....:                  'mixed gamma 0.25 '
   ....:                 , approximation='exact', log2=19, bs=1/4, normalize=False)
   ....:     er_table = np.linspace(.1, 2., 20)
   ....:     df = a01.density_df
   ....:     ix = df.index.get_indexer(er_table * a01.est_m, method='nearest')
   ....:     df = a01.density_df.iloc[ix][['loss', 'F', 'S', 'e', 'lev']]
   ....:     df['er'] = er_table
   ....:     df['charge'] = (df.e - df.lev) / a01.est_m
   ....:     df['savings'] = (df.loss - df.lev) / a01.est_m
   ....:     df['entry'] = df.loss / a01.est_m
   ....:     df = df.set_index('entry')
   ....:     df = df.drop(columns=['e',  'er'])
   ....:     df.index = [f"{x:.2f}" for x in df.index]
   ....:     df.index.name = 'r'
   ....:     Table = namedtuple('Table', ['ob', 'table_df'])
   ....:     return Table(a01, df)
   ....:
```

Finally, apply the new function to create some tables.

1. A small account with 25 expected claims, about 621K limited losses, and a low 50K occurrence limit. The output shows the usual `describe` diagnostics for the underlying `Aggregate` object, followed by a small Table across different entry ratios. The Table is indexed by entry ratio(aggregate attachment as a proportion of limited losses) and shows `loss` the aggregate limit loss level in currency units; the cdf and sf at that loss level (the latter giving the probability the aggregate layer attaches); the limited expected value at the entry ratio `lev`; and the insurance charge(`1 - lev / loss`) and savings (`r - lev / loss`).

```
In [16]: tl = make_table(25, 50)

In [17]: fc = lambda x: f'{x:,.1f}' if abs(x) > 10 else f'{x:.3f}'

In [18]: qd(tl.ob)

       E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
```

(continues on next page)

(continued from previous page)

```
Freq     25                        0.32016            0.51537
Sev   9.2527    9.2513 -0.00014797  1.7107     1.711   1.8407      1.8405
Agg   231.32    231.28 -0.00014797 0.46857   0.46862  0.65759     0.65759
log2 = 19, bandwidth = 1/4, validation: fails sev mean, agg mean.

In [19]: qd(tl.table_df, float_format=fc, col_space=8)

             loss        F        S      lev   charge  savings
r
0.10         23.2    0.005    0.995     23.2    0.900    0.000
0.20         46.2    0.017    0.983     46.0    0.801    0.001
0.30         69.5    0.041    0.959     68.6    0.703    0.004
0.40         92.5    0.081    0.919     90.2    0.610    0.010
0.50        115.8    0.135    0.865    111.0    0.520    0.021
0.60        138.8    0.205    0.795    130.1    0.438    0.037
0.70        162.0    0.284    0.716    147.7    0.361    0.062
0.80        185.0    0.370    0.630    163.2    0.294    0.094
0.90        208.2    0.459    0.541    176.8    0.236    0.136
1.00        231.2    0.544    0.456    188.3    0.186    0.186
1.10        254.5    0.625    0.375    197.9    0.144    0.245
1.20        277.5    0.696    0.304    205.7    0.110    0.310
1.30        300.8    0.760    0.240    212.0    0.083    0.384
1.40        323.8    0.813    0.187    216.9    0.062    0.462
1.50        347.0    0.857    0.143    220.8    0.045    0.546
1.60        370.0    0.892    0.108    223.7    0.033    0.633
1.70        393.2    0.920    0.080    225.8    0.024    0.724
1.80        416.2    0.941    0.059    227.4    0.017    0.816
1.90        439.5    0.958    0.042    228.6    0.012    0.912
2.00        462.5    0.970    0.030    229.4    0.008    1.008
```

2. The impact of increasing the occurrence limit to 250K:

```
In [20]: tl2 = make_table(25, 250)

In [21]: qd(tl2.ob)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     25                        0.32016            0.51537
Sev   16.989    16.988 -8.0799e-05  2.5894    2.5897    3.852       3.852
Agg   424.73    424.69 -8.0799e-05 0.60885   0.60889  0.91444     0.91445
log2 = 19, bandwidth = 1/4, validation: not unreasonable.

In [22]: qd(tl2.table_df, float_format=fc, col_space=8)

             loss        F        S      lev   charge  savings
r
0.10         42.5    0.015    0.985     42.3    0.900    0.001
0.20         85.0    0.052    0.948     83.4    0.804    0.004
0.30        127.5    0.104    0.896    122.7    0.711    0.011
0.40        170.0    0.164    0.836    159.5    0.624    0.025
0.50        212.2    0.227    0.773    193.5    0.544    0.044
0.60        254.8    0.291    0.709    225.0    0.470    0.070
0.70        297.2    0.358    0.642    253.7    0.403    0.102
0.80        339.8    0.428    0.572    279.6    0.342    0.142
0.90        382.2    0.497    0.503    302.4    0.288    0.188
1.00        424.8    0.562    0.438    322.4    0.241    0.241
1.10        467.2    0.622    0.378    339.7    0.200    0.300
1.20        509.8    0.677    0.323    354.6    0.165    0.365
1.30        552.0    0.726    0.274    367.2    0.135    0.435
1.40        594.5    0.770    0.230    377.9    0.110    0.510
```

(continues on next page)

(continued from previous page)

```
1.50        637.0    0.808    0.192    386.9    0.089    0.589
1.60        679.5    0.842    0.158    394.3    0.072    0.672
1.70        722.0    0.870    0.130    400.4    0.057    0.757
1.80        764.5    0.894    0.106    405.4    0.046    0.846
1.90        807.0    0.915    0.085    409.4    0.036    0.936
2.00        849.5    0.931    0.069    412.7    0.028    1.029
```

3. The impact of increasing the account size to 250 expected claims, still at 250K occurrence limit:

```
In [23]: tl3 = make_table(250, 250)

In [24]: qd(tl3.ob)

       E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq    250                         0.25788             0.50024
Sev   16.989   16.988 -8.0799e-05  2.5894     2.5897     3.852        3.852
Agg   4247.3   4246.9 -8.0799e-05 0.30548    0.30549   0.52614      0.52615
log2 = 19, bandwidth = 1/4, validation: not unreasonable.

In [25]: qd(tl3.table_df, float_format=fc, col_space=8)

             loss        F        S      lev   charge  savings
r
0.10        424.8    0.000    1.000    424.7    0.900    0.000
0.20        849.5    0.000    1.000    849.5    0.800    0.000
0.30      1,274.0    0.001    0.999  1,273.7    0.700    0.000
0.40      1,698.8    0.009    0.991  1,696.7    0.600    0.000
0.50      2,123.5    0.031    0.969  2,113.8    0.502    0.002
0.60      2,548.2    0.080    0.920  2,516.2    0.408    0.008
0.70      2,972.8    0.161    0.839  2,890.8    0.319    0.019
0.80      3,397.5    0.272    0.728  3,224.5    0.241    0.041
0.90      3,822.2    0.402    0.598  3,506.5    0.174    0.074
1.00      4,247.0    0.535    0.465  3,732.1    0.121    0.121
1.10      4,671.8    0.657    0.343  3,903.0    0.081    0.181
1.20      5,096.2    0.760    0.240  4,025.9    0.052    0.252
1.30      5,521.0    0.840    0.160  4,110.1    0.032    0.332
1.40      5,945.8    0.898    0.102  4,165.1    0.019    0.419
1.50      6,370.5    0.937    0.063  4,199.6    0.011    0.511
1.60      6,795.0    0.963    0.037  4,220.4    0.006    0.606
1.70      7,219.8    0.979    0.021  4,232.5    0.003    0.703
1.80      7,644.5    0.988    0.012  4,239.3    0.002    0.802
1.90      8,069.2    0.994    0.006  4,243.0    0.001    0.901
2.00      8,494.0    0.997    0.003  4,244.9    0.000    1.000
```

4. Finally, increase the occurrence limit to 10M:

```
In [26]: tl4 = make_table(250, 10000)

In [27]: qd(tl4.ob)

       E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq    250                         0.25788             0.50024
Sev    24.26   24.258 -5.6594e-05  6.2915     6.2918    33.759       33.759
Agg   6064.9   6064.6 -5.6594e-05 0.47416    0.47418    1.6385       1.6386
log2 = 19, bandwidth = 1/4, validation: not unreasonable.

In [28]: qd(tl4.table_df, float_format=fc, col_space=8)

             loss        F        S      lev   charge  savings
```

(continues on next page)

(continued from previous page)

```
r
0.10        606.5    0.000    1.000    606.5    0.900    0.000
0.20      1,213.0    0.001    0.999  1,212.9    0.800    0.000
0.30      1,819.2    0.008    0.992  1,817.1    0.700    0.000
0.40      2,425.8    0.033    0.967  2,412.5    0.602    0.002
0.50      3,032.2    0.086    0.914  2,984.7    0.508    0.008
0.60      3,638.8    0.168    0.832  3,515.7    0.420    0.020
0.70      4,245.2    0.272    0.728  3,989.4    0.342    0.042
0.80      4,851.8    0.385    0.615  4,396.7    0.275    0.075
0.90      5,458.0    0.496    0.504  4,735.5    0.219    0.119
1.00      6,064.5    0.595    0.405  5,010.4    0.174    0.174
1.10      6,671.0    0.680    0.320  5,229.3    0.138    0.238
1.20      7,277.5    0.750    0.250  5,401.4    0.109    0.309
1.30      7,884.0    0.805    0.195  5,535.8    0.087    0.387
1.40      8,490.5    0.847    0.153  5,640.8    0.070    0.470
1.50      9,096.8    0.880    0.120  5,723.1    0.056    0.556
1.60      9,703.2    0.905    0.095  5,788.1    0.046    0.646
1.70     10,309.8    0.924    0.076  5,839.8    0.037    0.737
1.80     10,916.2    0.938    0.062  5,881.6    0.030    0.830
1.90     11,522.8    0.949    0.051  5,915.7    0.025    0.925
2.00     12,129.0    0.958    0.042  5,943.8    0.020    1.020
```

These Tables all behave as expected. The insurance charge decreases with increasing expected losses (claim count) and decreasing occurrence limit.

2.5.3 Summary of Objects Created by DecL

Objects created by `build()` in this guide.

```
In [29]: from aggregate import pprint_ex

In [30]: for n, r in build.qlist('^IR:').iterrows():
   ....:     pprint_ex(r.program, split=20)
   ....:
```

2.6 Reinsurance Pricing

Objectives: Applications of the `Aggregate` class to reinsurance exposure rating, including swings and slides, aggregate stop loss and swing rated programs, illustrated using problems from CAS Parts 8 and 9.

Audience: Reinsurance pricing, broker, or ceded re actuary.

Prerequisites: DecL, underwriting and reinsurance terminology, aggregate distributions, risk measures.

See also: *The Reinsurance Clauses*, *Individual Risk Pricing*. For other related examples see *Published Problems and Examples*, especially *Bahnemann Monograph*.

Contents:

1. *Helpful references*
2. *Basic examples*
3. *Modes of reinsurance analysis*
4. *Reinsurance Functions*
5. *Casualty exposure rating*
6. *Property exposure rating*
7. *Variable features*

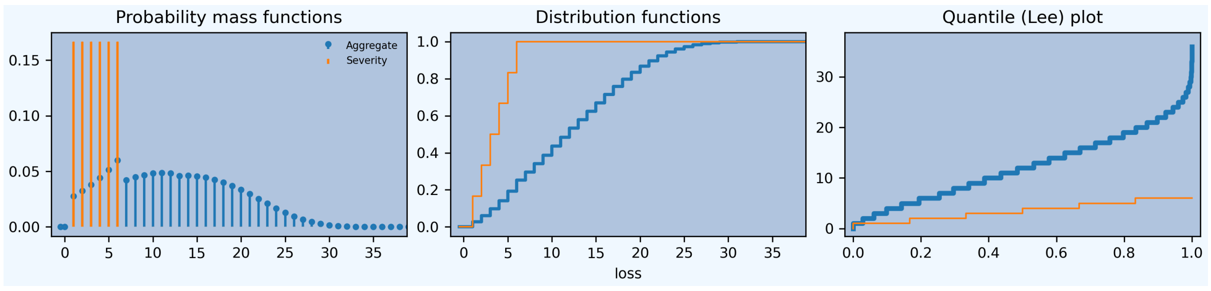
2.6.1 Helpful References

- General reinsurance: Strain [1997], Carter [2013], Albrecher *et al.* [2017], Parodi [2015]
- Clark [2014], CAS Part 8 reading
- General reinsurance pricing: Bear and Nemlick [1990], Mata *et al.* [2002]
- Property rating: Bernegger [1997], Ludwig [1991]
- Mildenhall and Major [2022] chapter 19

2.6.2 Basic Examples

Here are some basic examples. They are not realistic, but it is easy to see what is going on. The subsequent sections add realism. The basic example gross loss is a "die roll of dice rolls": roll a die, then roll that many dice and sum, see *Student*. The outcome is between 1 (probability 1/36) and 36 (probability 1/6**7), as confirmed by this output.

```
In [1]: import pandas as pd

In [2]: from aggregate import build, qd

In [3]: a01 = build('agg Re:01 '
   ...:             'dfreq [1 2 3 4 5 6] '
   ...:             'dsev [1 2 3 4 5 6] ')
   ...:

In [4]: a01.plot()

In [5]: qd(a01)

      E[X] Est E[X]   Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.5                     0.48795                  0
Sev    3.5      3.5          0 0.48795   0.48795        0  2.8529e-15
Agg  12.25    12.25 1.5543e-15 0.55328   0.55328  0.28689     0.28689
log2 = 7, bandwidth = 1, validation: not unreasonable.

In [6]: print(f'Pr D = 1:  {a01.pmf(1) : 11.6g} = {a01.pmf(1) * 36:.0f} / 36\n'
   ...:       f'Pr D = 36: {a01.pmf(36):8.6g} = {a01.pmf(36) * 6**7:.0f} / 6**7')
   ...:
Pr D = 1:    0.0277778 = 1 / 36
Pr D = 36: 3.57225e-06 = 1 / 6**7
```

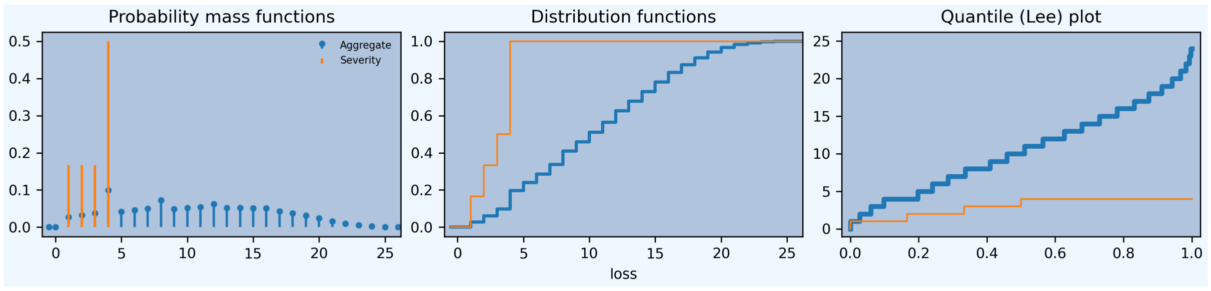


An **occurrence excess of loss** reinsurance layer is specified between the severity and frequency clauses because you need to know severity but not frequency. Multiple layers can be applied at once. This example enters 2 xs 4 as two layers:

```
occurrence net of 1 xs 4 and 1 xs 5
```

Requesting `net of` propagates losses net of the cover through to the aggregate.

```
In [7]: a02 = build('agg Re:02 '
   ...:             'dfreq [1:6] '
   ...:             'dsev [1:6] '
   ...:             'occurrence net of 1 xs 4 and 1 xs 5')
   ...:

In [8]: a02.plot()

In [9]: qd(a02)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.5                   0.48795                  0
Sev    3.5        3 -0.14286 0.48795    0.3849        0    -0.64952
Agg  12.25     10.5 -0.14286 0.55328   0.52955  0.28689     0.18324
log2 = 7, bandwidth = 1, validation: n/a, reinsurance.
```



`[1:6]` is shorthand for `[1,2,3,4,5,6]`. The net severity equals 3 = (1 + 2 + 3 + 4 + 4 + 4) / 6.

The `reinsurance_audit_df` dataframe shows unconditional (per ground up claim) severity statistics by layer. Multiply by the claim count `a02.n` to get layer loss picks. The severity, `ex`, equals (1 + 2) / 6 = 0.5 (first block). The expected loss to the layer equals 0.5 * 3.5 = 1.75 (second block).

```
In [10]: qd(a02.reinsurance_audit_df['ceded'])

                             ex     var      sd      cv    skew
kind share limit attach
occ  1.0   1.0   4.0    0.33333 0.22222  0.4714 1.4142 0.70711
                 5.0    0.16667 0.13889 0.37268 2.2361   1.7889
     all   inf   gup        0.5 0.58333 0.76376 1.5275   1.1223

In [11]: qd(a02.reinsurance_audit_df['ceded'], sparsify=False)

                             ex     var      sd      cv    skew
```

(continues on next page)

(continued from previous page)

```
kind share limit attach
occ  1.0   1.0   4.0    0.33333 0.22222  0.4714 1.4142 0.70711
occ  1.0   1.0   5.0    0.16667 0.13889 0.37268 2.2361  1.7889
occ  all   inf   gup        0.5 0.58333 0.76376 1.5275  1.1223

In [12]: qd(a02.reinsurance_audit_df['ceded'][['ex']] * a02.n)

                             ex
kind share limit attach
occ  1.0   1.0   4.0     1.1667
                 5.0    0.58333
     all   inf   gup       1.75
```

The `reinsurance_occ_layer_df` shows conditional layer expected loss and CV of loss, along with expected counts by layer and layer severity. The expected count to 1 xs 4 equals 3.5 / 3, because there is a 1/3 chance the layer attaches.

```
In [13]: qd(a02.reinsurance_occ_layer_df, sparsify=False)

stat                      ex     ex      ex     cv      cv      cv      en severity ␣
↪     pct
view                   ceded    net subject  ceded     net subject   ceded    ceded ␣
↪   ceded
share limit attach                                                                  ␣
↪
1.0   1.0   4.0      1.1667 11.083   12.25 1.4142 0.42433 0.48795  1.1667        1␣
↪0.095238
1.0   1.0   5.0     0.58333 11.667   12.25 2.2361 0.44721 0.48795 0.58333        1␣
↪0.047619
all   inf   gup        1.75   10.5   12.25 1.5275  0.3849 0.48795     3.5      0.5 ␣
↪0.14286
```

An **aggregate excess of loss** reinsurance layer, 12 xs 24, is specified after the frequency clause (you need to know frequency):

```
aggregate ceded to 12 xs 34.
```

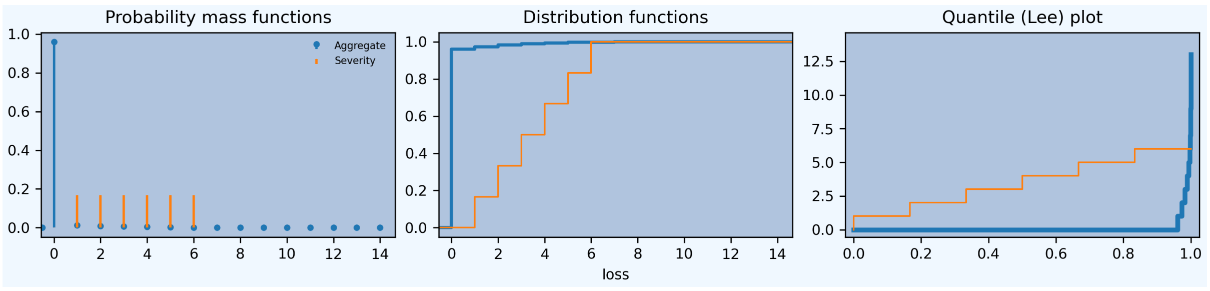
Requesting `ceded to` propagates the ceded losses through to the aggregate. Refer to `agg.Re:01` by name as a shorthand. `reinsurance_audit_df` reports expected loss to the aggregate layer. The layer is shown in two parts to illustrate reporting.

```
In [14]: a03 = build('agg Re:03 agg.Re:01 '
   ....:              'aggregate ceded to 6 xs 24 and 6 xs 30')
   ....:

In [15]: a03.plot()

In [16]: qd(a03)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.5                   0.48795                  0
Sev    3.5      3.5        0 0.48795   0.48795        0  2.8529e-15
Agg  12.25  0.10661  -0.9913 0.55328    5.9383  0.28689      7.6018
log2 = 7, bandwidth = 1, validation: n/a, reinsurance.

In [17]: qd(a03.reinsurance_audit_df.stack(0))

                                    ex       var       sd       cv     skew
kind share limit attach
agg  1.0   6.0   24.0   ceded     0.10378   0.36108   0.6009   5.7901   6.9432
```

(continues on next page)

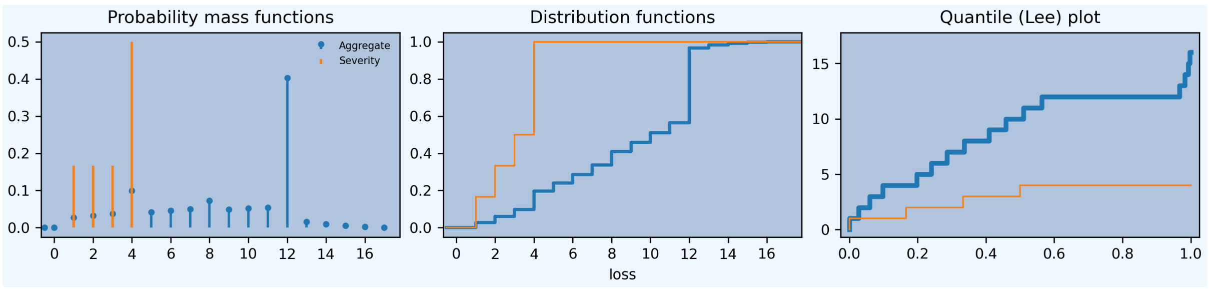
(continued from previous page)

```
                         net         12.146     43.082    6.5637 0.54039 0.15351
                         subject      12.25     45.938    6.7777 0.55328 0.28689
                  30.0   ceded    0.0028292 0.0063577 0.079735  28.183  35.705
                         net         12.247     45.831    6.7698 0.55277 0.27937
                         subject      12.25     45.938    6.7777 0.55328 0.28689
      all   inf   gup    ceded      0.10661     0.4008  0.63309  5.9383  7.6018
                         net         12.143     43.009    6.5581 0.54005  0.1501
                         subject      12.25     45.938    6.7777 0.55328 0.28689
```



Occurrence and aggregate programs can both be applied. The `ceded to` and `net of` clauses can be mixed. You cannot refer to `agg.Re:01` by name because you need to see into the object to apply the occurrence reinsurance.

```
In [18]: a04 = build('agg Re:04 dfreq [1:6] dsev [1:6] '
   ....:              'occurrence net of 1 xs 4 and 1 xs 5 '
   ....:              'aggregate net of 4 xs 12 and 4 xs 16')
   ....:

In [19]: a04.plot()

In [20]: qd(a04)

      E[X] Est E[X] Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.5                   0.48795                   0
Sev    3.5          3 -0.14286 0.48795    0.3849       0    -0.64952
Agg  12.25     8.8731 -0.27566 0.55328   0.41047 0.28689    -0.61588
log2 = 7, bandwidth = 1, validation: n/a, reinsurance.

In [21]: qd(a04.reinsurance_audit_df['ceded'])

                             ex      var       sd      cv    skew
kind share limit attach
occ  1.0   1.0   4.0    0.33333 0.22222   0.4714 1.4142 0.70711
                 5.0    0.16667 0.13889 0.37268 2.2361  1.7889
     all   inf   gup        0.5 0.58333 0.76376 1.5275  1.1223
agg  1.0   4.0   12.0     1.186  2.8219  1.6798 1.4164 0.88095
                 16.0   0.44087  1.1991    1.095 2.4838  2.4346
     all   inf   gup     1.6269  6.5022   2.5499 1.5674  1.3628
```



Layers can be specified as a **share of** or **part of** to account for coinsurance (partial placement) of the layer:

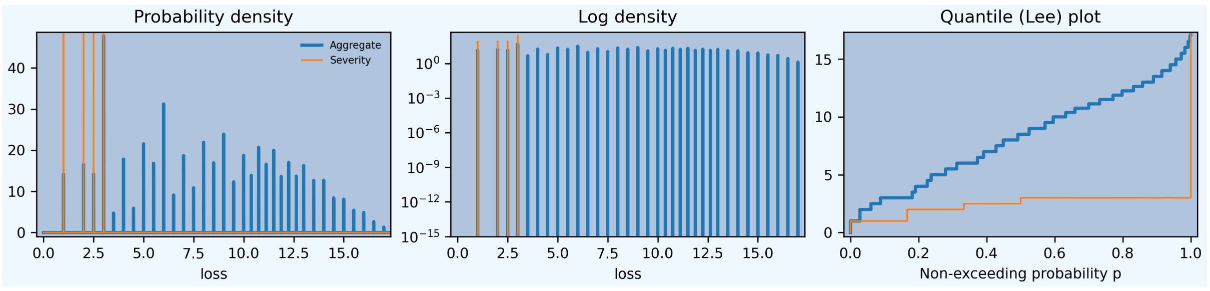
- `0.5 so 2 xs 2`, read 50% **share of** 2 xs 2, or

- `1 po 4 xs 10`, read 1 **part of** 4 xs 10.

> **Warning:** `aggregate` works with discrete distributions. All outcomes are multiples of the bucket size, `bs`. Any cession is rounded to a multiple of `bs`. Ensure `bs` is appropriate to capture cessions when applying share or part of. By default `build` uses `bs=1` when it detects a discrete distribution, such as the die roll example. Ceding to `0.5 so 2 xs 2` produces ceded losses of 0.5 and net losses of 2.5. To capture these needs a much smaller discretization grid. Non-discrete aggregates plot as though they are continuous or mixed distributions.

These concepts are illustrated in the next example. Note the bucket size.

```
In [22]: a05 = build('agg Re:05 '
   ....:              'dfreq [1:6] dsev [1:6] '
   ....:              'occurrence net of 0.5 so 2 xs 2 and 2 xs 4 '
   ....:              'aggregate net of 1 po 4 xs 10 '
   ....:              , bs=1/512, log2=16)
   ....:

In [23]: a05.plot()

In [24]: qd(a05)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.5                   0.48795                  0
Sev    3.5   2.4167 -0.30952 0.48795   0.30258        0    -0.98865
Agg  12.25   8.2063  -0.3301 0.55328   0.49122  0.28689   0.0035585
log2 = 16, bandwidth = 1/512, validation: n/a, reinsurance.

In [25]: qd(a05.reinsurance_audit_df['ceded'])

                              ex     var      sd      cv     skew
kind share limit attach
occ  0.5   2.0   2.0     0.58333 0.20139 0.44876 0.76931 -0.33297
     1.0   2.0   4.0         0.5 0.58333 0.76376  1.5275   1.1223
     all   inf   gup      1.0833  1.2014  1.0961  1.0118  0.64163
agg  0.25  4.0   10.0    0.25202 0.14516 0.38099  1.5117   1.1168
     all   inf   gup     0.25202 0.14516 0.38099  1.5117   1.1168
```



A **tower** of limits can be specified by giving the attachment points of each layer. The shorthand:

```
occurrence ceded to tower [0 1 2 5 10 20 36]
```

is equivalent to:

```
occurrence ceded to 1 xs 0 and 1 xs 1 and 3 xs 2 \
and 5 xs 5 and 10 xs 10 and 16 xs 20
```

Here is a summary of these examples. The audit dataframe gives a layering of aggregate losses. The plot is omitted; it is identical to gross since the tower covers all losses.

```
In [26]: a06 = build('agg Re:06 '
   ....:             'agg.Re:01 '
   ....:             'aggregate ceded to tower [0 1 2 5 10 20 36]')
   ....:

In [27]: a06.plot()

In [28]: qd(a06)

      E[X] Est E[X]   Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.5                     0.48795                  0
Sev    3.5      3.5          0 0.48795   0.48795        0  2.8529e-15
Agg  12.25    12.25 2.2204e-16 0.55328   0.55328  0.28689     0.28689
log2 = 7, bandwidth = 1, validation: n/a, reinsurance.

In [29]: qd(a06.reinsurance_audit_df['ceded'], sparsify=False)

                             ex        var         sd         cv         skew
kind share limit attach
agg  1.0   1.0   0.0          1 4.4409e-16 2.1073e-08 2.1073e-08 -4.7453e+07
agg  1.0   1.0   1.0    0.97222   0.027006    0.16434    0.16903       -5.747
agg  1.0   3.0   2.0     2.6997    0.64684    0.80426    0.29791      -2.6177
agg  1.0   5.0   5.0     3.5291     4.2422     2.0597    0.58361     -0.87928
agg  1.0   10.0  10.0     3.573     15.609     3.9508     1.1057      0.56831
agg  1.0   16.0  20.0   0.47593     2.2625     1.5042     3.1604       3.8783
agg  all   inf   gup      12.25     45.938     6.7777    0.55328      0.28689
```

See *Reinsurance Functions* for more about the reinsurance functions.

2.6.3 Modes of Reinsurance Analysis

Inwards reinsurance pricing is begins with an estimated loss pick, possibly supplemented by distribution and volatility statistics such as loss standard deviation or quantiles. `aggregate` can help in two ways.

1. Excess of loss exposure rating that accounts for the limits profile of the underlying business and how it interacts with excess layers. Uses only the severity distribution through difference of increased limits factors. This application is peripheral to the underlying purpose of `aggregate`, but is very convenient nonetheless.
2. The impact of treaty **variable features** that are derived from the full aggregate distribution of ceded losses and expenses—a showcase application.

Outwards reinsurance is evaluated based on the loss pick and the impact of the cession on the distribution of retained losses. Ceded re and broker actuaries often want the full gross and net outcome distributions.

2.6.4 Reinsurance Functions

This section demonstrates `Aggregate` methods and properties for reinsurance analysis. These are:

- `reinsurance_kinds()` a text description of the kinds (occurrence and/or aggregate) of reinsurance applied.
- `reinsurance_description()` a text description of the layers and shares, by kind.
- `reinsurance_occ_plot()` plots subject (usually gross), ceded, and net severity, and aggregates created from each. Does not consider aggregate reinsurance.
- `reinsurance_audit_df` dataframe summary by ceded, net, and subject, showing mean, CV, SD, and skewness of occurrence loss by layer and in total by kind.
- `reinsurance_occ_layer_df` dataframe showing an expected loss layering analysis for occurrence reinsurance.

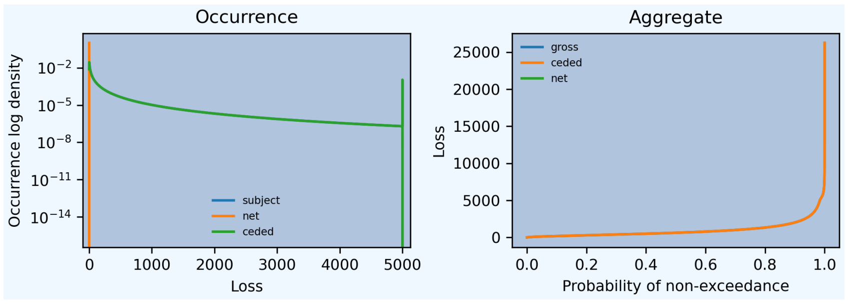
- `reinsurance_df` dataframe showing all possible densities.
- `reinsurance_report_df` dataframe showing mean, CV, skew, and SD statistics for each column in `reinsurance_df`.

These are illustrated using the a more realistic example that includes occurrence and aggregate reinsurance. Notice that the occurrence program just layers gross (subject) losses. Gross losses are then passed through to the aggregate program. This is done to illustrate the functions below. In a real-world application is is likely the bottom few occurrence layers would be dropped and you would pass the net of through to the aggregate.

```
In [30]: from aggregate import build, qd

In [31]: a = build('agg ReTester '
   ....:           '10 claims '
   ....:           '5000 xs 0 '
   ....:           'sev lognorm 100 cv 5 '
   ....:           'occurrence ceded to 250 xs 0 and 250 xs 250 and 500 xs 500 and␣
↪1000 xs 1000 and 3000 xs 2000 '
   ....:           'poisson '
   ....:           'aggregate ceded to 250 xs 750 and 1500 xs 1000 '
   ....:          )
   ....:

In [32]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     10                      0.31623            0.31623
Sev   95.058    95.057 -2.0186e-06  3.2307    3.2307   9.4329      9.4329
Agg   950.58     310.2    -0.67367  1.0695    1.7183   2.8646      1.7399
log2 = 16, bandwidth = 1/2, validation: n/a, reinsurance.

In [33]: print(a.reinsurance_kinds())
Occurrence and aggregate

In [34]: print(a.reinsurance_description())
Ceded to 100% share of 250 xs 0 and 100% share of 250 xs 250 and 100% share of 500␣
↪xs 500 and 100% share of 1,000 xs 1,000 and 100% share of 3,000 xs 2,000 per␣
↪occurrence then ceded to 100% share of 250 xs 750 and 100% share of 1,500 xs 1,
↪000 in the aggregate.
```

`'plot` shows the impact of occurrence reinsurance on severity and aggregate losses, and the ceded severity and aggregate.

```
In [35]: a.reinsurance_occ_plot()
```



The `reinsurance_audit_df` dataframe shows unconditional layer severity that "adds-up" to the total layer severity; compare to the total with the severity statistics in description above. These only match when the reinsurance layers exhaust the ground-up limit.

```
In [36]: qd(a.reinsurance_audit_df, sparsify=False)
```

(continues on next page)

(continued from previous page)

```
                          ceded      ceded   ceded   ceded    ceded     net          net␣
↪   net      net   \
                              ex          var      sd      cv     skew      ex          var␣
↪    sd       cv
kind share limit  attach                                                                   ␣
↪
occ  1.0    250.0  0.0     54.459      5603.2 74.854 1.3745   1.6845 40.599       72833␣
↪269.88  6.6474
occ  1.0    250.0  250.0   13.305      2720.7   52.16 3.9204   3.9945 81.753       73469␣
↪271.05  3.3155
occ  1.0    500.0  500.0   11.481      4748.2 68.907 6.0018   6.3905 83.576       64191␣
↪253.36  3.0315
occ  1.0    1000.0 1000.0 8.7092      7140.4 84.501 9.7025   10.609 86.348       57052␣
↪238.86  2.7662
occ  1.0    3000.0 2000.0 7.1035       15770 125.58 17.678   20.596 87.954       51379␣
↪226.67  2.5771
occ  all    inf     gup    95.057       94314 307.11 3.2307   9.4329       0           0␣
↪     0      NaN
agg  1.0    250.0  750.0   89.854       13133  114.6 1.2754 0.58829 860.72 8.8125e+05␣
↪938.75  1.0907
agg  1.0    1500.0 1000.0 220.35  2.004e+05 447.67 2.0316   2.0217 730.22 4.2122e+05␣
↪649.02 0.88879
agg  all    inf     gup      310.2 2.8411e+05 533.02 1.7183   1.7399 640.37 3.3956e+05␣
↪582.72 0.90997

                            net  subject    subject  subject  subject  subject
                           skew       ex        var       sd       cv     skew
kind share limit  attach
occ  1.0    250.0  0.0     11.537   95.057      94314   307.11   3.2307   9.4329
occ  1.0    250.0  250.0    10.96   95.057      94314   307.11   3.2307   9.4329
occ  1.0    500.0  500.0    10.74   95.057      94314   307.11   3.2307   9.4329
occ  1.0    1000.0 1000.0 8.7905   95.057      94314   307.11   3.2307   9.4329
occ  1.0    3000.0 2000.0 5.6194   95.057      94314   307.11   3.2307   9.4329
occ  all    inf     gup       NaN   95.057      94314   307.11   3.2307   9.4329
agg  1.0    250.0  750.0    3.242   950.57 1.0335e+06   1016.6   1.0695   2.8646
agg  1.0    1500.0 1000.0 3.9477   950.57 1.0335e+06   1016.6   1.0695   2.8646
agg  all    inf     gup    4.8085   950.57 1.0335e+06   1016.6   1.0695   2.8646
```

The `reinsurance_occ_layer_df` dataframe shows unconditional aggregate statistics. The blocks `ex` and `cv` show values from `audit_df` times expected claim counts; `en` shows claim counts by layer. `severity` shows the implied conditional layer severity, equal to expected loss from `audit_df` divided by the probability of attaching the layer.

```
In [37]: qd(a.reinsurance_occ_layer_df, sparsify=False)

stat                       ex      ex       ex      cv      cv       cv        en  severity ␣
↪    pct
view                    ceded     net  subject  ceded     net  subject     ceded     ceded ␣
↪  ceded
share limit  attach                                                                         ␣
↪
1.0   250.0  0.0     544.59 405.99   950.57 1.3745 6.6474   3.2307        10    54.459 ␣
↪ 0.5729
1.0   250.0  250.0   133.05 817.53   950.57 3.9204 3.3155   3.2307   0.79248    167.89 ␣
↪0.13997
1.0   500.0  500.0   114.81 835.76   950.57 6.0018 3.0315   3.2307   0.36394    315.47 ␣
↪0.12078
1.0   1000.0 1000.0 87.092 863.48   950.57 9.7025 2.7662   3.2307   0.14697    592.59 ␣
↪0.09162
1.0   3000.0 2000.0 71.035 879.54   950.57 17.678 2.5771   3.2307 0.052009    1365.8␣
↪0.074729
```

(continues on next page)

(continued from previous page)

```
all    inf     gup     950.57       0   950.57  3.2307     NaN   3.2307        10    95.057 ␣
↪       1
```

The `reinsurance_df` density dataframe shows subject, ceded, and net occurrence (severity); aggregates created from each (without aggregate reinsurance); and subject, ceded, and net of requested aggregate reinsurance.

```
In [38]: qd(a.reinsurance_df, max_rows=20)

         loss   p_sev_gross   p_sev_ceded   p_sev_net   p_agg_gross_occ   p_agg_ceded_
↪occ   \
loss                                                                                   ␣
↪
0.0         0     0.0078283     0.0078283           1        4.9097e-05        4.9097e-
↪05
0.5       0.5      0.027461      0.027461           0        1.3482e-05        1.3482e-
↪05
1.0         1      0.028318      0.028318           0        1.5754e-05        1.5754e-
↪05
1.5       1.5      0.026716      0.026716           0        1.7104e-05        1.7104e-
↪05
2.0         2      0.024836      0.024836           0          1.83e-05          1.83e-
↪05
2.5       2.5      0.023058      0.023058           0        1.9467e-05        1.9467e-
↪05
3.0         3      0.021453      0.021453           0        2.0635e-05        2.0635e-
↪05
3.5       3.5      0.020023      0.020023           0        2.1812e-05        2.1812e-
↪05
4.0         4       0.01875       0.01875           0           2.3e-05           2.3e-
↪05
4.5       4.5      0.017615      0.017615           0          2.42e-05          2.42e-
↪05
...       ...           ...           ...         ...               ...            ..
↪.
32763.0 32763             0             0           0                 0              ␣
↪0
32763.5 32764             0             0           0                 0              ␣
↪0
32764.0 32764             0             0           0                 0              ␣
↪0
32764.5 32764             0             0           0                 0              ␣
↪0
32765.0 32765             0             0           0                 0              ␣
↪0
32765.5 32766             0             0           0                 0              ␣
↪0
32766.0 32766             0             0           0                 0              ␣
↪0
32766.5 32766             0             0           0                 0              ␣
↪0
32767.0 32767             0             0           0                 0              ␣
↪0
32767.5 32768             0             0           0                 0              ␣
↪0

         p_agg_net_occ  p_agg_gross  p_agg_ceded  p_agg_net
loss
0.0                  1   4.9097e-05      0.57612 4.9097e-05
0.5                  0   1.3482e-05   0.00029446 1.3482e-05
1.0                  0   1.5754e-05   0.00029422 1.5754e-05
1.5                  0   1.7104e-05   0.00029398 1.7104e-05
```

(continues on next page)

(continued from previous page)

```
2.0                    0     1.83e-05   0.00029374   1.83e-05
2.5                    0   1.9467e-05   0.00029351 1.9467e-05
3.0                    0   2.0635e-05   0.00029327 2.0635e-05
3.5                    0   2.1812e-05   0.00029303 2.1812e-05
4.0                    0      2.3e-05   0.00029279    2.3e-05
4.5                    0     2.42e-05   0.00029256   2.42e-05
...                  ...          ...          ...        ...
32763.0                0   4.3423e-19            0          0
32763.5                0   4.2049e-19            0          0
32764.0                0   4.3521e-19            0          0
32764.5                0   4.2965e-19            0          0
32765.0                0   4.2249e-19            0          0
32765.5                0   4.1451e-19            0          0
32766.0                0   4.2417e-19            0          0
32766.5                0   4.0984e-19            0          0
32767.0                0   4.2856e-19            0          0
32767.5                0   3.9884e-19            0          0
```

The `reinsurance_report_df` shows statistics for the densities in `reinsurance_df`. The `p_agg_gross` column matches the theoretical (gross) output shown in `qd(a)` at the top and the `p_agg_ceded` column matches the estimated output because the aggregate program requested `ceded to` output. The net column is the difference.

```
In [39]: qd(a.reinsurance_report_df)

      p_sev_gross  p_sev_ceded  p_sev_net  p_agg_gross_occ  p_agg_ceded_occ  p_agg_
↪net_occ  \
mean        95.057       95.057          0           950.57           950.57       ␣
↪       0
cv          3.2307       3.2307        NaN           1.0695           1.0695       ␣
↪     NaN
sd          307.11       307.11        NaN           1016.6           1016.6       ␣
↪     NaN
skew        9.4329       9.4329        NaN           2.8646           2.8646       ␣
↪     NaN

      p_agg_gross  p_agg_ceded  p_agg_net
mean       950.57        310.2     640.37
cv         1.0695       1.7183    0.90997
sd         1016.6       533.02     582.72
skew       2.8646       1.7399     4.8085
```

2.6.5 Casualty Exposure Rating

This example calculates the loss pick for excess layers across a subject portfolio with different underlying limits and deductibles but a common severity curve. The limit profile is given by a premium distribution and the expected loss ratio varies by limit. Values are in 000s. Policies at 1M and 2M limits are ground-up and those at 5M and 10M limits have a 100K and 250K deductible. The full assumptions are:

```
In [40]: profile = pd.DataFrame({'limit': [1000, 2000, 5000, 10000],
   ....:                         'ded'  : [0, 0, 100, 250],
   ....:                         'premium': [10000, 5000, 2500, 1500],
   ....:                         'lr': [.75, .75, .7, .65]
   ....:                        }, index=pd.Index(range(4), name='class'))
   ....:

In [41]: qd(profile)

       limit  ded  premium   lr
class
```

(continues on next page)

(continued from previous page)

```
0       1000     0    10000 0.75
1       2000     0     5000 0.75
2       5000   100     2500  0.7
3      10000   250     1500 0.65
```

The severity is a lognormal with an unlimited mean of 50 and cv of 10, $\sigma = 2.148$. The gross portfolio and tower are created in `a07`. A typical XOL tower up to 10M is created by specifying the layer break points in an `occurrence ceded to tower` clause.

```
In [42]: a07 = build('agg Re:07 '
   ....:             f'{profile.premium.values} premium at {profile.lr.values} lr '
   ....:             f'{profile.limit.values} xs {profile.ded.values} '
   ....:              'sev lognorm 50 cv 10 '
   ....:              'occurrence ceded to tower [0 250 500 1000 2000 5000 10000] '
   ....:              'poisson '
   ....:              , approximation='exact', log2=18, bs=1/2)
   ....:

In [43]: qd(a07)

       E[X] Est E[X]     Err E[X]     CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 292.72                         0.058448           0.058448
Sev   47.741    47.737 -8.6055e-05    4.0096      4.01    15.901      15.901
Agg    13975     13974 -8.6055e-05  0.24153   0.24156  0.88974     0.88974
log2 = 18, bandwidth = 1/2, validation: n/a, reinsurance.
```

There are special options in `build` because the claim count is high: 292.7. To force a convolution use `approximation='exact'`. Reviewing the default `bs=1/2` and `log2=16` shows a moderate error. Looking at the density via:

```
a07.density_df.p_total.plot(logy=True)
```

shows aliasing, i.e., there is not enough space in the answer. Adjust by increasing `log2` from 16 to 18 and leaving `bs=1/2`.

The dataframe `reinsurance_occ_layer_df` shows layer expected loss, CV, counts, and conditional severity. The last column shows the percent of subject ceded to each layer.

```
In [44]: qd(a07.reinsurance_occ_layer_df, sparsify=False)

stat                    ex          ex       ex      cv      cv       cv        en␣
↪severity       pct
view                 ceded         net  subject  ceded     net  subject     ceded   ␣
↪ceded      ceded
share limit   attach                                                                 ␣
↪
1.0    250.0   0.0     8724.1      5249.7    13974 1.9905 8.8564     4.01    292.72   29.
↪803    0.62432
1.0    250.0   250.0   2076.6       11897    13974 5.4793 4.0236     4.01    12.063  ␣
↪172.15    0.14861
1.0    500.0   500.0   1917.9       12056    13974 8.0564 3.7267     4.01    5.8545  ␣
↪327.58    0.13725
1.0    1000.0 1000.0 775.37        13198    13974 17.887 3.5805     4.01    1.2306   ␣
↪630.1   0.055488
1.0    3000.0 2000.0 400.74        13573    13974 41.327  3.509     4.01   0.26392  ␣
↪1518.4   0.028678
1.0    5000.0 5000.0  79.06        13895    13974 122.64 3.8196     4.01 0.027983  ␣
↪2825.3 0.0056577
all    inf     gup     13974 -7.8694e-09    13974   4.01     NaN     4.01    292.72   47.
↪737          1
```

2.6.6 Property Risk Exposure Rating

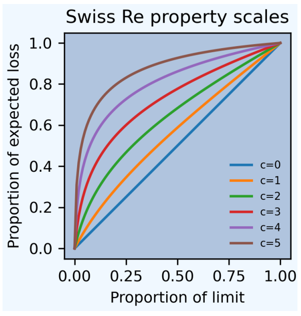
Property risk exposure rating differs from casualty in part because the severity distribution varies with each risk (location). Rather than a single ground-up severity curve per class, there is a size of loss distribution normalized by property total insured value (TIV).

We start by introducing the Swiss Re severity curves, Bernegger [1997] using a moments-matched beta distribution. The function `G` defines the MBBEFD distribution, parameterized by `c`.

```
In [45]: from aggregate import xsden_to_meancv

In [46]: import scipy.stats as ss

In [47]: import numpy as np

In [48]: import matplotlib.pyplot as plt

In [49]: def bb(c):
   ....:     return np.exp(3.1 - 0.15*c*(1+c))
   ....:

In [50]: def bg(c):
   ....:     return np.exp((0.78 + 0.12*c)*c)
   ....:

In [51]: def G(x, c):
   ....:     b = bb(c)
   ....:     g = bg(c)
   ....:     return np.log(((g - 1) * b + (1 - g * b) * b**x) / (1 - b)) / np.
↪log(g * b)
   ....:
```

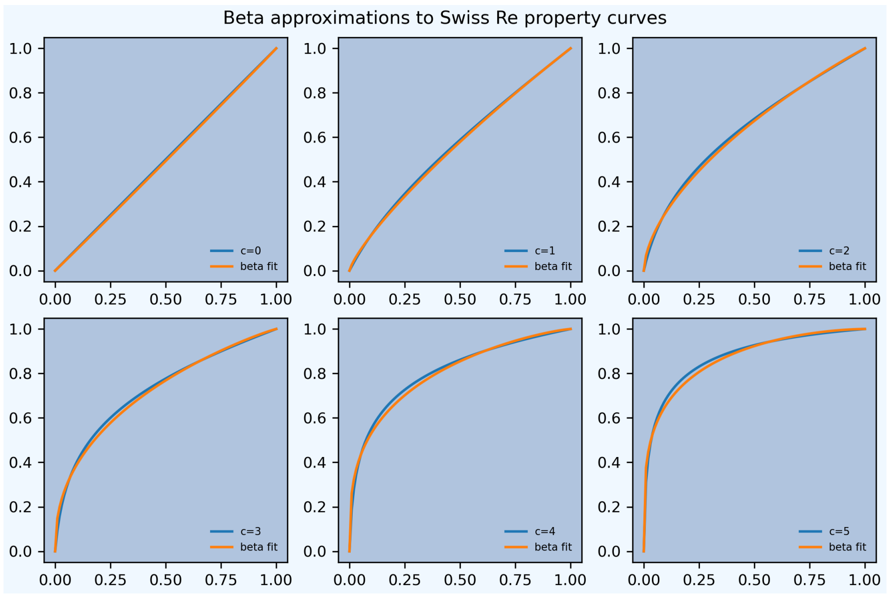
Here are the base curves, compare Figure 4.2 in Bernegger [1997]. The curve `c=5` is close to the Lloyd's curve (scale).

```
In [52]: fig, ax = plt.subplots(1, 1, figsize=(2.45, 2.55), constrained_
↪layout=True)

In [53]: ans = []

In [54]: ps = np.linspace(0,1,101)

In [55]: for c in [0, 1, 2, 3, 4, 5]:
   ....:     gs = G(ps, c)
   ....:     ax.plot(ps, gs, label=f'c={c}')
   ....:     ans.append([c, *xsden_to_meancv(ps[1:], np.diff(gs))])
   ....:

In [56]: ax.legend(loc='lower right');

In [57]: ax.set(xlabel='Proportion of limit', ylabel='Proportion of expected loss',
   ....:        title='Swiss Re property scales');
   ....:
```



Next, approximate these curves with a beta distribution to make them easier for us to use in `aggregate`. Here are the parameters and fit graphs for each curve.

```
In [58]: swiss = pd.DataFrame(ans, columns=['c', 'mean', 'cv'])

In [59]: def beta_ab(m, cv):
   ....:     v = (m * cv) ** 2
   ....:     sev_a = m * (m * (1 - m) / v - 1)
   ....:     sev_b = (1 - m) * (m * (1 - m) / v - 1)
   ....:     return sev_a, sev_b
   ....:

In [60]: a, b = beta_ab(swiss['mean'], swiss.cv)

In [61]: swiss['a'] = a

In [62]: swiss['b'] = b

In [63]: swiss = swiss.set_index('c')

In [64]: qd(swiss)

     mean       cv        a       b
c
0   0.505  0.57161     1.01    0.99
1 0.44108  0.67278  0.79375  1.0058
2 0.36415  0.81654  0.58953  1.0294
3 0.28003   1.0103  0.42538  1.0937
4 0.19858   1.2531  0.31176  1.2582
5 0.13101   1.5171  0.24654  1.6353

In [65]: fig, axs = plt.subplots(2, 3, figsize=(3 * 2.45, 2 * 2.45), constrained_
↪layout=True)

In [66]: for ax, (c, r) in zip(axs.flat, swiss.iterrows()):
   ....:     gs = G(ps, c)
   ....:     fz = ss.beta(r.a, r.b)
   ....:     ax.plot(ps, gs, label=f'c={c}')
   ....:     ax.plot(ps, fz.cdf(ps), label=f'beta fit')
   ....:     ans.append([c, *xsden_to_meancv(ps[1:], np.diff(gs))])
   ....:     ax.legend(loc='lower right');
   ....:

In [67]: fig.suptitle('Beta approximations to Swiss Re property curves');
```



Work on a property schedule with the following TIVs and deductibles. The premium rate is 0.35 per 100 and the loss ratio is 55%.

```
In [68]: schedule = pd.DataFrame({
   ....:     'locid': range(10),
   ....:     'tiv': [850, 950, 1250, 1500, 4500, 8000, 9000, 12000, 25000, 50000],
   ....:     'ded': [ 10,  10,   20,   20,   50,   100, 500,  1000,  5000,  5000]}
   ....:     ).set_index('locid')
   ....:

In [69]: schedule['premium'] = schedule.tiv / 100 * 0.35

In [70]: schedule['lr'] = 0.55

In [71]: qd(schedule)

          tiv   ded  premium   lr
locid
0         850    10    2.975 0.55
1         950    10    3.325 0.55
2        1250    20    4.375 0.55
3        1500    20     5.25 0.55
4        4500    50    15.75 0.55
5        8000   100       28 0.55
6        9000   500     31.5 0.55
7       12000  1000       42 0.55
8       25000  5000     87.5 0.55
9       50000  5000      175 0.55
```

Build the stochastic model using a Swiss Re `c=3` scale. Use a gamma mixed Poisson frequency with a CV of 3 to reflect the potential for catastrophe losses. Use a `tower` clause to set up the analysis of a per risk tower. Increase `bs` to 2 based on high error with recommended `bs=1`.

```
In [72]: beta_a, beta_b = swiss.loc[3, ['a', 'b']]

In [73]: a08 = build('agg Re:08 '
   ....:              f'{schedule.premium.values} premium at {schedule.lr.values} lr
↪'
   ....:              f'{schedule.tiv.values} xs {schedule.ded.values} '
   ....:              f'sev {schedule.tiv.values} * beta {beta_a} {beta_b} ! '
   ....:               'occurrence ceded to tower [0 1000 5000 10000 20000 inf] '
```

(continues on next page)

(continued from previous page)

```
   ....:              'mixed gamma 2 '
   ....:              , bs=2)
   ....:

In [74]: qd(a08)

         E[X] Est E[X]    Err E[X]  CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 0.081693                        4.03             5.0226
Sev     2663.9   2663.9 -1.2094e-07 2.2311    2.2311   3.8455      3.8455
Agg     217.62   217.62 -1.3778e-07 8.7849    8.7849   14.303      14.303
log2 = 16, bandwidth = 2, validation: n/a, reinsurance.
```

The shared mixing increases the frequency and aggregate CV and skewness.

```
In [75]: qd(a08.report_df.loc[
   ....:     ['freq_m', 'freq_cv', 'freq_skew', 'agg_cv', 'agg_skew'],
   ....:     ['independent', 'mixed']])
   ....:

view      independent    mixed
statistic
freq_m       0.081693 0.081693
freq_cv        3.5576     4.03
freq_skew      3.6749   5.0226
agg_cv         8.6169   8.7849
agg_skew       14.248   14.303
```

Look at `reinsurance_occ_layer_df` to summarize the analysis.

```
In [76]: qd(a08.reinsurance_occ_layer_df, sparsify=False)

stat                          ex      ex       ex       cv      cv       cv         en␣
↪severity     pct
view                       ceded     net  subject    ceded     net  subject      ceded   ␣
↪ceded   ceded
share limit   attach                                                                    ␣
↪
1.0   1000.0  0.0      38.473 179.15   217.62 0.93539   2.612   2.2311   0.05879   654.
↪41 0.17679
1.0   4000.0  1000.0  74.293 143.33   217.62   1.6859 2.7968   2.2311  0.027946  ␣
↪2658.4 0.34139
1.0   5000.0  5000.0  43.156 174.47   217.62   2.6872 2.2324   2.2311  0.012623  ␣
↪3418.8 0.19831
1.0   10000.0 10000.0 37.782 179.84   217.62   4.1476 1.9578   2.2311 0.0059027  ␣
↪6400.9 0.17362
1.0   inf     20000.0 23.917  193.7   217.62   7.2389   1.91   2.2311 0.0021465   ␣
↪11143  0.1099
all   inf     gup     217.62       0   217.62   2.2311     NaN   2.2311  0.081693  ␣
↪2663.9       1
```

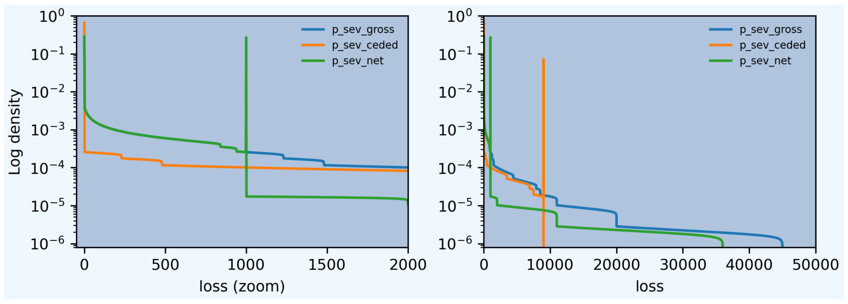
Add plots of gross, ceded, and net severity with the placed program, 4000 xs 1000 and 5000 xs 5000. (The net is zero with the `tower` clause, so we have to recompute.) The left and right plots differ only in the x-axis scale.

```
In [77]: a09 = build('agg Re:09 '
   ....:             f'{schedule.premium.values} premium at {schedule.lr.values} lr
↪'
   ....:             f'{schedule.tiv.values} xs {schedule.ded.values} '
   ....:             f'sev {schedule.tiv.values} * beta {beta_a} {beta_b} ! '
   ....:              'occurrence ceded to 4000 xs 1000 and 5000 xs 5000 '
   ....:              'mixed gamma 2 ', bs=2)
```

(continues on next page)

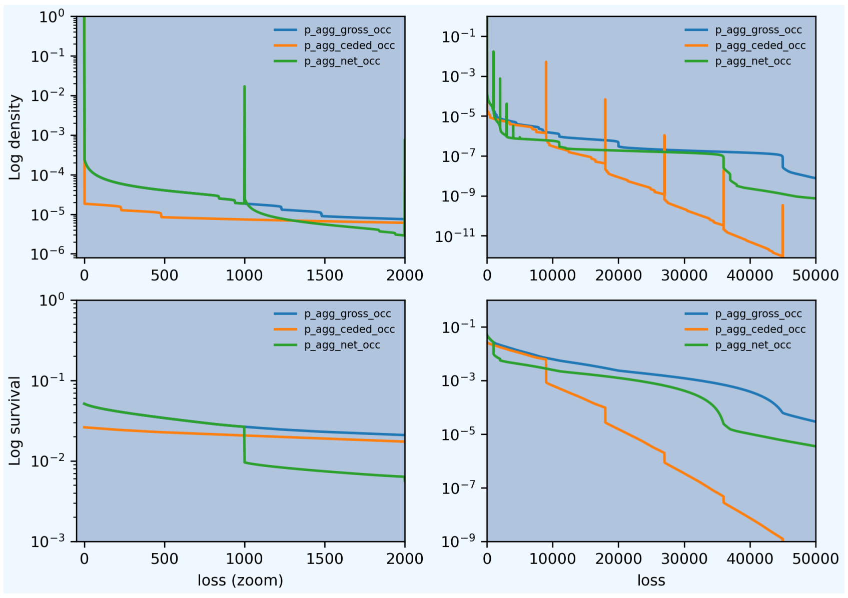
(continued from previous page)

```
   ....:

In [78]: qd(a09)

          E[X] Est E[X] Err E[X]  CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 0.081693                     4.03              5.0226
Sev     2663.9   1437.7 -0.46031 2.2311    1.9215   3.8455      1.8785
Agg     217.62   117.45 -0.46031 8.7849     7.838   14.303      9.4024
log2 = 16, bandwidth = 2, validation: n/a, reinsurance.

In [79]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True); \
   ....: ax0, ax1 = axs.flat; \
   ....: df = a09.reinsurance_df; \
   ....: df.filter(regex='sev_[gcn]').plot(logy=True, xlim=[-50, 2000], ylim=[0.8e-
↪6, 1] , ax=ax0); \
   ....: df.filter(regex='sev_[gcn]').plot(logy=True, xlim=[0, 50000], ylim=[0.8e-
↪6, 1], ax=ax1); \
   ....: ax0.set(xlabel='loss (zoom)', ylabel='Log density');
   ....:

In [80]: ax1.set(xlabel='loss', ylabel='');
```



And finally, plot the corresponding aggregate distributions.

```
In [81]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.45), constrained_
↪layout=True); \
   ....: ax0, ax1, ax2, ax3 = axs.flat; \
   ....: df.filter(regex='agg_.*_occ').plot(logy=True, xlim=[-50, 2000], ylim=[0.
↪8e-6, 1] , ax=ax0); \
   ....: (1 - df.filter(regex='agg_.*_occ').cumsum()).plot(logy=True, xlim=[-50,␣
↪2000], ylim=[1e-3, 1], ax=ax2); \
   ....: df.filter(regex='agg_.*_occ').plot(logy=True, xlim=[0, 50000], ylim=[0.8e-
↪12, 1], ax=ax1); \
   ....: (1 - df.filter(regex='agg_.*_occ').cumsum()).plot(logy=True, xlim=[0,␣
↪50000], ylim=[1e-9, 1], ax=ax3); \
   ....: ax0.set(xlabel='', ylabel='Log density'); \
   ....: ax1.set(xlabel='', ylabel=''); \
   ....: ax2.set(xlabel='loss (zoom)', ylabel='Log survival');
   ....:

In [82]: ax3.set(xlabel='loss', ylabel='');
```



2.6.7 Variable Features

Reinsurance treaties can incorporate **variable features** that alter the contract cash flows. These can impact losses, premiums, or expenses (through the ceding commission). They can apply to quota share and excess treaties.

- Variable features altering **Loss** cash flows
  - Aggregate limits and deductibles
  - Loss corridor
  - Limited reinstatements for excess treaties, by number of covered events or an aggregate limit
- Variable features altering **Premium** cash flows
  - Swing or retro rating or margin-plus premium, where the premium equals losses times an expense factor subject to a maximum and minimum. See also *Individual Risk Pricing*.
- Variable features altering **Expense** cash flows
  - Sliding scale commission
  - Profit commission or profit share

A loss corridor and sliding scale commission have a similar impact; both concentrate the impact of the treaty on tail outcomes. Aggregate features have the opposite effect; concentrating the impact on body outcomes and lowering effectiveness on tail outcomes.

Premium and expense related features are substitutes, the former used on treaties without ceding commissions.

2.6.8 Inwards Analysis of Bear and Nemlick Variable Features

Bear and Nemlick [1990] analyze six treaties with variable features across four portfolios. These examples are included because they are realistic and show that `aggregate` produces the same answers as a published reference. The subject losses defined as follows.

- **Treaty 1 and 4.**
  - Cover: 160 xs 40
  - Subject business
    - ∗ Two classes
    - ∗ Subject premium 3000 and 6000

    - Loss rate 4% and 3%
    - Severity: single parameter Pareto with shape 0.9 and 0.95

- **Treaty 2 and 5.**
  - Cover: 700 xs 300
  - Subject business
    - Three classes
    - Subject premium 2000 each
    - Loss rate 10%, 14%, 21%
    - Severity: single parameter Pareto with shape 1.5, 1.3, 1.1
- **Treaty 3.**
  - Cover: 400 xs 100
  - Subject business
    - Three classes
    - Subject premium 4500, 4500, 1000
    - Loss rate 3.2%, 3.8%, 3.5%
    - Severity: single parameter Pareto with shape 1.1.
- **Treaty 6.**
  - Cover: 900 xs 100
  - Subject business
    - Subject premium 25000
    - Layer loss cost 10% of subject premium
    - Portfolio CV 0.485

They include a variety of frequency assumptions, including Poisson, negative binomial with variance multiplier based on a gross multiplier of 2 or 3 adjusted for excess frequency, mixing variance 0.05 and 0.10. Excess counts get closer to Poisson and so the difference between the two is slight.

The next table shows Bear and Nemlick's estimated premium rates.

Heckman and Meyers describe claim count contagion and frequency parameter uncertainty, which they model using a mixed-Poisson frequency distribution. Their parameter c is the variance of the mixing distribution. The value `c=0.05` is replicated in DecL with the frequency clause `mixed gamma 0.05**0.5`, since DecL is based on the CV of the mixing distribution (the mean is always 1).

Heckman and Meyers also describe severity parameter uncertainty, which they model with an inverse gamma variable with mean 1 and variance b. There is no analog of severity uncertainty in DecL. For finite excess layers it has a muted impact on results. Heckman and Meyers call c the contagion parameter and b the mixing parameter, which is confusing in our context. To approximate these columns use

- `c=0,b=0` corresponds to the DecL frequency clause `poisson`.
- `c=0.05,b=...` is close to DecL frequency clause `mixed gamma 0.05**0.5`.
- `c=0.1,b=...` is close to DecL frequency clause `mixed gamma 0.1**0.5`.

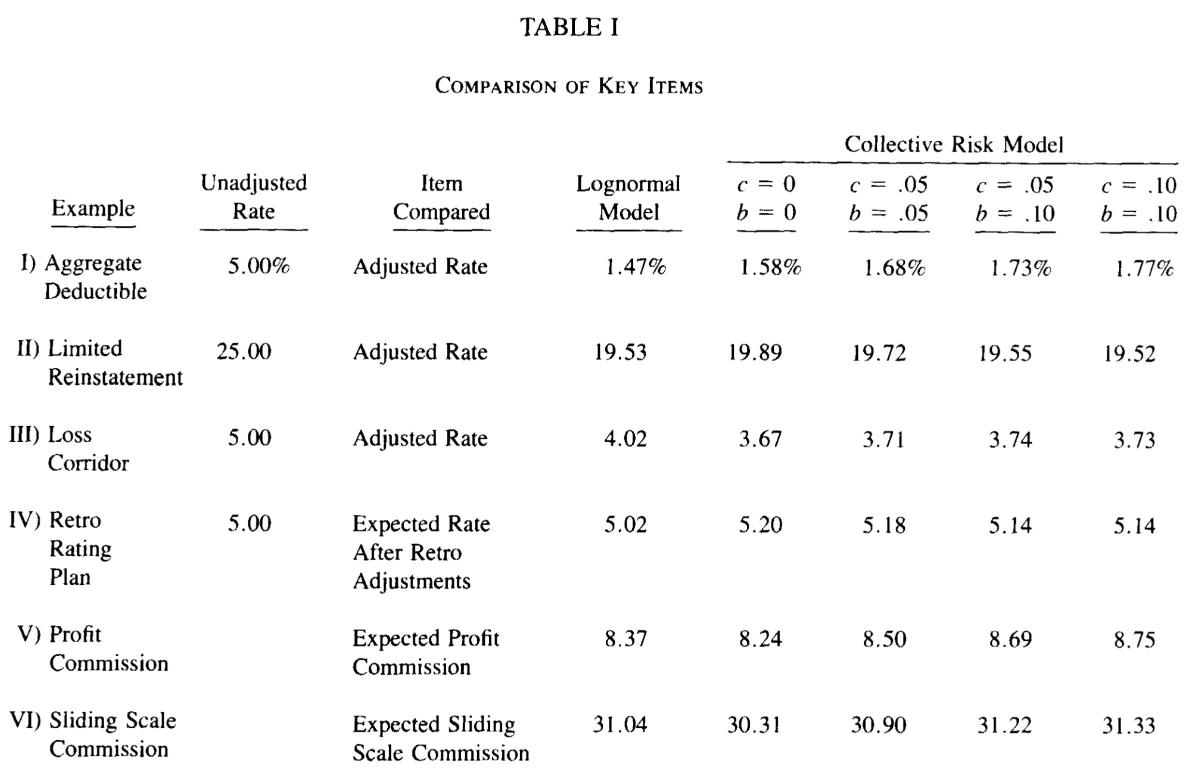


TABLE I

COMPARISON OF KEY ITEMS

				Collective Risk Model			
Example	Unadjusted Rate	Item Compared	Lognormal Model	$c = 0$ $b = 0$	$c = .05$ $b = .05$	$c = .05$ $b = .10$	$c = .10$ $b = .10$
I) Aggregate Deductible	5.00%	Adjusted Rate	1.47%	1.58%	1.68%	1.73%	1.77%
II) Limited Reinstatement	25.00	Adjusted Rate	19.53	19.89	19.72	19.55	19.52
III) Loss Corridor	5.00	Adjusted Rate	4.02	3.67	3.71	3.74	3.73
IV) Retro Rating Plan	5.00	Expected Rate After Retro Adjustments	5.02	5.20	5.18	5.14	5.14
V) Profit Commission		Expected Profit Commission	8.37	8.24	8.50	8.69	8.75
VI) Sliding Scale Commission		Expected Sliding Scale Commission	31.04	30.31	30.90	31.22	31.33

Fig. 1: Bear and Nemlick's estimated premium rates by program by numerical method. The Lognormal Model column uses a method of moments fit to the aggregate mean and CV. The Collective Risk Model columns uses the Heckman-Meyers continuous FFT method.

Specifying the Single Parameter Pareto

Losses to an excess layer specified by a single parameter Pareto are the same as losses to a ground-up layer with a shifted Pareto.

Example.

For 400 xs 100 and Pareto shape 1.1, these two DecL programs produce identical results:

```
4 claims 400 xs 100 sev 100 * pareto 1.1 poisson

4 claims 400 xs 0 sev 100 * pareto 1.1 - 100 poisson
```

Treaty 1: Aggregate Deductible

Treaty 1 adds an aggregate deductible of 360, equal to 3% of subject premium.

Setup the gross portfolio.

```
In [83]: import numpy as np

In [84]: from aggregate import build, mv, qd, xsden_to_meancvskew, \
   ....:      mu_sigma_from_mean_cv, lognorm_lev
   ....:

In [85]: mix_cv = ((1.036-1)/5.154)**.5; mix_cv
Out[85]: 0.08357551150546018

In [86]: a10 = build('agg Re:BN1 '
```

(continues on next page)

(continued from previous page)

```
   ....:              '[9000 3000] exposure at [0.04 0.03] rate '
   ....:              '160 xs 0 '
   ....:              'sev 40 * pareto [0.9 0.95] - 40 '
   ....:             f'mixed gamma {mix_cv} ')
   ....:

In [87]: qd(a10)

       E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq 6.4966                        0.40114             0.41855
Sev   69.267    69.267 -1.2751e-08 0.87703   0.87703  0.50056     0.50056
Agg      450       450 -2.1418e-07  0.5285    0.5285  0.62441     0.62436
log2 = 16, bandwidth = 1/32, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.
```

The portfolio CV matches 0.528, reported in Bear and Nemlick Appendix F, Exhibit 1.

There are several ways to estimate the impact of the AAD on recovered losses.

By hand, adjust losses and use the distribution of outcomes from `a.density_df`. The last line computes the sum-product of losses net of AAD times probabilities, i.e., the expected loss cost.

```
In [88]: bit = a10.density_df[['loss', 'p_total']]

In [89]: bit['loss'] = np.maximum(0, bit.loss - 360)

In [90]: bit.prod(axis=1).sum()
Out[90]: 142.7571229766539
```

More in the spirit of `aggregate`: create a new `Aggregate` applying the AAD using a DecL `aggregate net of` reinsurance clause. Alternatively use `aggregate ceded to inf xs 360` (not shown).

```
In [91]: a11 = build('agg Re:BN1a '
   ....:              '[9000 3000] exposure at [0.04 0.03] rate '
   ....:              '160 xs 0 '
   ....:              'sev 40 * pareto [0.9 0.95] - 40 '
   ....:             f'mixed gamma {mix_cv} '
   ....:              'aggregate net of 360 xs 0 ')
   ....:

In [92]: qd(a11)

       E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq 6.4966                        0.40114             0.41855
Sev   69.267    69.267 -1.2751e-08 0.87703   0.87703  0.50056     0.50056
Agg      450    142.76    -0.68276  0.5285    1.2873  0.62441      1.5337
log2 = 16, bandwidth = 1/32, validation: n/a, reinsurance.

In [93]: gross = a11.agg_m; net = a11.est_m; ins_charge = net / gross

In [94]: net, ins_charge
Out[94]: (142.75888602897527, 0.3172419689532783)
```

Bear and Nemlick use a lognormal approximation to the aggregate.

```
In [95]: mu, sigma = mu_sigma_from_mean_cv(a10.agg_m, a10.agg_cv)

In [96]: elim_approx = lognorm_lev(mu, sigma, 1, 360)
```

(continues on next page)

(continued from previous page)

```
In [97]: a11.agg_m - elim_approx, 1 - elim_approx / a11.agg_m
Out[97]: (132.06333218858032, 0.29347407153017846)
```

The lognormal overstates the value of the AAD, resulting in a lower net premium. This is because the approximating lognormal is much more skewed.

```
In [98]: fz = a11.approximate('lognorm')

In [99]: fz.stats('s'), a11.est_skew
Out[99]: (5.99543353838776, 1.5337492513585025)
```

Bear and Nemlick report the Poisson approximation and a Heckman-Meyers convolution with mixing and contagion equal 0.05. We can compute the Poisson exactly and approximate Heckman-Meyers with contagion but no mixing. Changing 0.05 to 0.10 is close to the `b=0.1` column.

```
In [100]: a12 = build('agg Re:BN1p '
   .....:             '[9000 3000] exposure at [0.04 0.03] rate '
   .....:             '160 xs 0 '
   .....:             'sev 40 * pareto [0.9 0.95] - 40 '
   .....:            f'poisson '
   .....:             'aggregate net of 360 xs 0 ')
   .....:

In [101]: qd(a12)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 6.4966                      0.39234            0.39234
Sev   69.267   69.267 -1.2751e-08 0.87703   0.87703  0.50056     0.50056
Agg      450    141.8    -0.68489 0.52185    1.2794  0.60775      1.5152
log2 = 16, bandwidth = 1/32, validation: n/a, reinsurance.

In [102]: a13 = build('agg Re:BN1c '
   .....:             '[9000 3000] exposure at [0.04 0.03] rate '
   .....:             '160 xs 0 '
   .....:             'sev 40 * pareto [0.9 0.95] - 40 '
   .....:             'mixed gamma 0.05**.5 '
   .....:             'aggregate net of 360 xs 0 ')
   .....:

In [103]: qd(a13)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 6.4966                      0.45158             0.5623
Sev   69.267   69.267 -5.1006e-08 0.87703   0.87703  0.50056     0.50056
Agg      450   148.41     -0.6702 0.56774    1.3332  0.72251      1.6412
log2 = 16, bandwidth = 1/16, validation: n/a, reinsurance.
```

Here is a summary of the different methods, compare Bear and Nemlick Table 1, row 1, page 75.

```
In [104]: bit = pd.DataFrame([a10.agg_m,
   .....:     a11.describe.iloc[-1, 1],
   .....:     a12.describe.iloc[-1, 1],
   .....:     a13.describe.iloc[-1, 1],
   .....:     a11.agg_m - elim_approx],
   .....:     columns=['Loss cost'],
   .....:     index=pd.Index(['Gross', 'NB', 'Poisson', 'c=0.05', 'lognorm'],
   .....:                    name='Method'))
   .....:
```

(continues on next page)

(continued from previous page)

```
In [105]: bit['Premium'] = bit['Loss cost'] * 100 / 75

In [106]: bit['Rate'] = bit.Premium / 12000

In [107]: qd(bit, accuracy=5)

         Loss cost  Premium     Rate
Method
Gross          450      600     0.05
NB          142.76   190.35 0.015862
Poisson      141.8   189.07 0.015755
c=0.05      148.41   197.88  0.01649
lognorm     132.06   176.08 0.014674
```

Treaty 2: Aggregate Limit

Treaty 2 adds an aggregate limit of 2800, i.e., 3 full reinstatements plus the original limit.

Setup the gross portfolio.

```
In [108]: a14 = build('agg Re:BN2 '
   .....:             '[2000 2000 2000] exposure at [.1 .14 .21] rate '
   .....:             '700 xs 0 '
   .....:             'sev 300 * pareto [1.5 1.3 1.1] - 300 '
   .....:             'mixed gamma 0.07 '
   .....:             , bs=1/8)
   .....:

In [109]: qd(a14)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 2.8949                       0.5919            0.60017
Sev   310.9    310.9 -8.2604e-09 0.83714   0.83714  0.46335     0.46335
Agg     900      900 -8.4265e-09 0.76969   0.76969  0.90207     0.90207
log2 = 16, bandwidth = 1/8, validation: not unreasonable.
```

Specify `bs=1/8` since the error was too high with the default `bs=1/16`. The portfolio CV matches 0.770, reported in Bear and Nemlick Appendix G, Exhibit 1. The easiest way to value the aggregate limit to use an `aggregate ceded to` clause.

```
In [110]: a14n = build('agg Re:BN2a '
   .....:              '[2000 2000 2000] exposure at [.1 .14 .21] rate '
   .....:              '700 xs 0 '
   .....:              'sev 300 * pareto [1.5 1.3 1.1] - 300 '
   .....:              'mixed gamma 0.07 '
   .....:              'aggregate ceded to 2800 xs 0'
   .....:              , bs=1/8)
   .....:

In [111]: qd(a14n)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 2.8949                       0.5919            0.60017
Sev   310.9    310.9 -8.2604e-09 0.83714   0.83714  0.46335     0.46335
Agg     900   894.68  -0.0059104 0.76969    0.7544  0.90207     0.72143
log2 = 16, bandwidth = 1/8, validation: n/a, reinsurance.
```

Applying a 20% coinsurance and grossing up by 100/60 produces the premium and rate. Using Poisson frequency, or mixed gamma with mix $\sqrt{0.05}$ or $\sqrt{0.1}$ ties closely to Table I, row 2.

```
In [112]: p = a14n.est_m * (1 - 0.2) * 100 / 60

In [113]: p, p / 6000
Out[113]: (1192.9075706627575, 0.19881792844379292)
```

`aggregate` induces correlation between the three classes because they share mixing variables. The `report_df` shows the details by line and compares with an independent sum.

```
In [114]: qd(a14.report_df.iloc[:, :-2])

view              0       1       2 independent   mixed
statistic
name         Re:BN2  Re:BN2  Re:BN2      Re:BN2  Re:BN2
limit           700     700     700         700     700
attachment        0       0       0           0       0
el              200     280     420         900     900
freq_m      0.73701 0.92362  1.2342      2.8949  2.8949
freq_cv      1.1669  1.0429 0.90284     0.58919  0.5919
freq_skew    1.1711  1.0476 0.90827     0.59209 0.60017
sev_m        271.37  303.15   340.3       310.9   310.9
sev_cv       0.9087 0.84973 0.78339     0.83714 0.83714
sev_skew    0.72529 0.51332 0.27989     0.46335 0.46335
agg_m           200     280     420         900     900
agg_cv       1.5755  1.3672  1.1456     0.76767 0.76969
agg_skew     1.9025  1.6073  1.3121     0.89708 0.90207
```

Treaty 3: Loss Corridor

Treaty 3 is a loss corridor from expected layer losses to twice expected. The reinsurance pays up to expected and beyond twice expected.

Setup the gross portfolio with CV 0.905. Use a larger `bs` to reduce error.

```
In [115]: a15 = build('agg Re:BN3 '
   .....:             '[4500 4500 1000] exposure at [.032 .038 .035] rate '
   .....:             '400 xs 0 '
   .....:             'sev 100 * pareto 1.1 - 100 '
   .....:             'poisson', bs=1/16)
   .....:

In [116]: qd(a15)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 2.3544                      0.65172             0.65172
Sev  148.66   148.66 -1.1633e-08 0.96405   0.96405  0.79141     0.79141
Agg     350      350 -1.1694e-08 0.90526   0.90526   1.0937      1.0937
log2 = 16, bandwidth = 1/16, validation: not unreasonable.
```

There are several ways to model a loss corridor, but the most natural is to use an `aggregate net of 350 xs 350` clause; expected layer loss equals 350.

```
In [117]: a15_lc = build('agg Re:BN3lc '
   .....:                '[4500 4500 1000] exposure at [.032 .038 .035] rate '
   .....:                '400 xs 0 '
   .....:                'sev 100 * pareto 1.1 - 100 '
   .....:                'poisson '
   .....:                'aggregate net of 350 xs 350 ', bs=1/16)
   .....:
```

(continues on next page)

(continued from previous page)

```
In [118]: qd(a15_lc)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 2.3544                      0.65172            0.65172
Sev  148.66   148.66 -1.1633e-08 0.96405   0.96405  0.79141     0.79141
Agg     350   256.88    -0.26607 0.90526   0.79669   1.0937      1.2854
log2 = 16, bandwidth = 1/16, validation: n/a, reinsurance.
```

Compare the results with the lognormal approximation, see Table 1 line 3.

```
In [119]: mu, sigma = mu_sigma_from_mean_cv(1, 0.905)

In [120]: ler = lognorm_lev(mu, sigma, 1, 2) - lognorm_lev(mu, sigma, 1, 1)

In [121]: p = a15_lc.est_m * 100 / 70

In [122]: bit = pd.DataFrame(
   .....:     [a15_lc.est_m, 1 - a15_lc.est_m / a15.est_m, ler, p, p/10000,
   .....:      350 * (1 - ler) * 100 / 70 / 10000, 350 * 100 / 70 / 10000],
   .....:     index=pd.Index(['Loss cost', 'LER', 'Lognorm LER', 'Premium',
   .....:                    'Rate', 'Lognorm rate', 'Unadjusted rate'],name='Item
↪'),
   .....:     columns=['Value'])
   .....:

In [123]: qd(bit, accuracy=4)

                    Value
Item
Loss cost          256.88
LER               0.26607
Lognorm LER       0.19535
Premium            366.97
Rate             0.036697
Lognorm rate     0.040233
Unadjusted rate      0.05
```

Treaty 4: Retro Rated Program

Treaty 4 is a retro rated program on the same business as Treaty 1. The flat rate is 5%, given by a 100/75 load on the 3.75% loss cost. Subject premium equals 12000. The retrospective rating plan has a one-year adjustment period. The adjusted treaty premium equals 100/75 times incurred losses and ALAE in the layer limited to a maximum of 10% of subject premium and a minimum of 3%.

The gross portfolio is the same as Treaty 1. Use Poisson frequency.

```
In [124]: a16 = build('agg Re:BN4 '
   .....:             '[9000 3000] exposure at [0.04 0.03] rate '
   .....:             '160 xs 0 '
   .....:             'sev 40 * pareto [0.9 0.95] - 40 '
   .....:             'poisson ')
   .....:

In [125]: qd(a16)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 6.4966                      0.39234            0.39234
```

(continues on next page)

(continued from previous page)

```
Sev   69.267    69.267 -1.2751e-08 0.87703    0.87703  0.50056      0.50056
Agg      450       450 -1.3429e-07 0.52185    0.52185  0.60775      0.60771
log2 = 16, bandwidth = 1/32, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.
```

The estimated retro premium (`erp`) and corresponding rate are easy to compute.

```
In [126]: bit = a16.density_df[['loss', 'p_total']]

In [127]: subject = 12000; min_rate = 0.03; max_rate = 0.10; lcf = 100 / 75

In [128]: bit['premium'] = np.minimum(max_rate * subject,
   .....:                             np.maximum(min_rate * subject, lcf * bit.
↪loss))
   .....:

In [129]: erp = bit[['premium', 'p_total']].prod(1).sum()

In [130]: erp, erp / subject
Out[130]: (624.508953754649, 0.05204241281288742)
```

Bear and Nemlick also report the lognormal approximation.

```
In [131]: from scipy.integrate import quad

In [132]: fz = a16.approximate('lognorm')

In [133]: lognorm_approx = quad(lambda x: min(max_rate * subject,
   .....:                          max(min_rate * subject, lcf * x)) * fz.pdf(x),
   .....:                         0, np.inf)
   .....:

In [134]: lognorm_approx[0], lognorm_approx[0] / subject
Out[134]: (602.6887696334271, 0.05022406413611893)
```

Treaty 5: Profit Share

Treaty 5 models a three-year profit commission on the business underlying Treaty 2. The three years are modeled independently with no change in exposure, giving 18M subject premium. The terms of the profit commission are a 25% share after a 20% expense allowance (“25% after 20%”), meaning a profit share payment equal to 25% of the “profit” to the reinsurer after losses and a 20% expense allowance.

The treaty rate equals 25% of subject premium. There is a 20% proportional coinsurance that does not correspond to an explicit share of the reinsurance premium (i.e., the 25% rate is for 80% cover). The analysis of Treaty 2 shows the loss cost equals 900, or a 15% rate.

The ceded loss ratio equals (loss rate) x (coinsurance) / (premium rate) = 0.15 * 0.8 / 0.25 = 0.48.

The profit commission formula is:

```
pc = 0.25 * max(0, 1 - (loss ratio) - 0.2) * (subject premium)
   = 0.25 * max(0, premium * 0.8 - loss).
```

The expected profit commission rate, ignoring Jensen’s inequality, equals:

```
pc rate = 0.25 * (1 - 0.48 - 0.2) = 0.25 * 0.32 = 0.08.
```

We can compute the expected value across the range of outcomes from the aggregate distribution.

Use a `Portfolio` object to aggregate the three years. It is convenient to create the single year distribution and then use the `Underwriter` to refer to it by name.

```
In [135]: a17 = build('agg Re:BN2p '
   .....:             '[2000 2000 2000] exposure at [.1 .14 .21] rate '
   .....:             '700 xs 0 '
   .....:             'sev 300 * pareto [1.5 1.3 1.1] - 300 '
   .....:             'poisson')
   .....:

In [136]: p17 = build('port Treaty.5 '
   .....:             'agg Year.1 agg.Re:BN2p '
   .....:             'agg Year.2 agg.Re:BN2p '
   .....:             'agg Year.3 agg.Re:BN2p '
   .....:             , bs=1/4)
   .....:

In [137]: qd(p17)

              E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
unit   X
Year.1 Freq 2.8949                       0.58774             0.58774
       Sev   310.9    310.9 -3.3041e-08 0.83714    0.83714  0.46335      0.46335
       Agg     900      900 -3.3041e-08  0.7665     0.7665  0.89409      0.89409
Year.2 Freq 2.8949                       0.58774             0.58774
       Sev   310.9    310.9 -3.3041e-08 0.83714    0.83714  0.46335      0.46335
       Agg     900      900 -3.3041e-08  0.7665     0.7665  0.89409      0.89409
Year.3 Freq 2.8949                       0.58774             0.58774
       Sev   310.9    310.9 -3.3041e-08 0.83714    0.83714  0.46335      0.46335
       Agg     900      900 -3.3041e-08  0.7665     0.7665  0.89409      0.89409
total  Freq 8.6846                       0.33933             0.33933
       Sev   310.9    310.9 -3.3041e-08 0.83714             0.46335
       Agg    2700     2700 -3.3043e-08 0.44254    0.44254   0.5162       0.5162
log2 = 16, bandwidth = 1/4, validation: not unreasonable.
```

The three-year total CV equals 0.443 with Poisson frequency. Bear and Nemlick Appendix J, Exhibit 2, shows 0.444 with negative binomial frequency.

Compute the estimated profit share payment by hand.

```
In [138]: subject_premium = 18000; coinsurance = 0.2; re_rate = 0.25

In [139]: pc_share = 0.25; pc_expense = 0.2

In [140]: bit = p17.density_df[['loss', 'p_total']]

In [141]: bit['lr'] = bit.loss * (1 - coinsurance) / (re_rate * subject_premium)

In [142]: bit['pc_rate'] = np.maximum(0, pc_share * (1 - pc_expense - bit.lr))

In [143]: pc_pmt = (bit.pc_rate * bit.p_total).sum()

In [144]: print(f'Estimated pc payment rate = {pc_pmt:.4g}')
Estimated pc payment rate = 0.08238
```

Table 1 shows a rate of 8.24% for Poisson frequency.

Exercise. Replicate the rate computed using a lognormal approximation and a negative binomial `c=0.05`. Reconcile to Table 1.

Note. If the premium varies by year then the builtin object can be scaled. There are two ways to scale aggregate distributions.

1. **Homogeneous** scaling, using `*` to scale severity;
2. **Inhomogeneous** scaling, using `@` to scale expected frequency and exposure.

See Mildenhall [2004] and Mildenhall [2017] for an explanation of why homogeneous scaling is appropriate for assets whereas inhomogeneous scaling applies to insurance. See Boonen *et al.* [2017] for an application.

Here is an extreme example to illustrate the differences. Homogeneous scaling does not change the aggregate CV or skewness (or any other scaled higher moment or the shape of the distribution). Inhomogeneous scaling changes the shape of the distribution; it becomes more symmetric, decreasing the CV and skewness.

```
In [145]: p17growing = build('port Treaty.5 '
   .....:                    'agg Year.1 agg.Re:BN2p '
   .....:                    'agg Year.2 2 @ agg.Re:BN2p '
   .....:                    'agg Year.3 2 * agg.Re:BN2p '
   .....:                    , bs=1/4)
   .....:

In [146]: qd(p17growing)

              E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit   X
Year.1 Freq 2.8949                      0.58774            0.58774
       Sev   310.9    310.9 -3.3041e-08 0.83714   0.83714  0.46335     0.46335
       Agg     900      900 -3.3041e-08  0.7665    0.7665  0.89409     0.89409
Year.2 Freq 5.7897                       0.4156             0.4156
       Sev   310.9    310.9 -3.3041e-08 0.83714   0.83714  0.46335     0.46335
       Agg    1800     1800 -3.3041e-08   0.542     0.542  0.63222     0.63222
Year.3 Freq 2.8949                      0.58774            0.58774
       Sev  621.79   621.79 -8.2604e-09 0.83714   0.83714  0.46335     0.46335
       Agg    1800     1800 -8.3769e-09  0.7665    0.7665  0.89409     0.89409
total  Freq 11.579                      0.29387            0.29387
       Sev  388.62   388.62 -2.3129e-08 0.95127             1.2038
       Agg    4500     4500 -3.6822e-06  0.4056   0.40559  0.53104     0.53077
log2 = 16, bandwidth = 1/4, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.
```

Note. The following DecL program will produce the same answer as the `Portfolio` called `p17` above. The exposure has been tripled.

```
In [147]: a17p= build('agg Re:BN6p '
   .....:             '[6000 6000 6000] exposure at [.1 .14 .21] rate '
   .....:             '700 xs 0 '
   .....:             'sev 300 * pareto [1.5 1.3 1.1] - 300 '
   .....:             'poisson'
   .....:             , bs=1/4)
   .....:

In [148]: qd(a17p)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 8.6846                     0.33933            0.33933
Sev   310.9    310.9 -3.3041e-08 0.83714   0.83714  0.46335     0.46335
Agg    2700     2700 -3.3041e-08 0.44254   0.44254   0.5162      0.5162
log2 = 16, bandwidth = 1/4, validation: not unreasonable.
```

However, for a mixed frequency the answers are different, because mixing is shared mixing across class and year, producing a higher CV and skewness.

```
In [149]: a17nb = build('agg Re:BN6c '
   .....:               '[6000 6000 6000] exposure at [.1 .14 .21] rate '
   .....:               '700 xs 0 '
   .....:               'sev 300 * pareto [1.5 1.3 1.1] - 300 '
   .....:               'mixed gamma 0.1**.5'
   .....:               , bs=1/4)
```

(continues on next page)

(continued from previous page)

```
   .....:

In [150]: qd(a17nb)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 8.6846                      0.46384            0.67943
Sev   310.9    310.9 -3.3041e-08 0.83714   0.83714  0.46335     0.46335
Agg    2700     2700 -3.7443e-08 0.54391   0.54391  0.76744     0.76744
log2 = 16, bandwidth = 1/4, validation: not unreasonable.
```

Treaty 6: Sliding Scale Commission

Treaty 6 models a one-year sliding scale commission plan. The details of the plan are:

- Minimum commission of 20% at or above a 65% loss ratio
- Slide 0.5:1 between 55% and 65% to a 25% commission
- Slide 0.75:1 between 35% and 55% to a 40% commission
- Maximum commission of 40% at or below a 35% loss ratio.

The underlying portfolio is specified only as a 900 xs 100 layer on 25M premium with a 10% layer loss cost and a CV of 0.485. No other details are provided. Based on trial and error and the other examples, model the portfolio using a single parameter Pareto with $q = 1.05$ and a gamma mixed Poisson with mixing CV 0.095.

```
In [151]: a18 = build('agg Re:BN5 '
   .....:             '25000 exposure at 0.1 rate '
   .....:             '900 xs 0 '
   .....:             'sev 100 * pareto 1.05 - 100 '
   .....:             'mixed gamma 0.095')
   .....:

In [152]: qd(a18)

       E[X] Est E[X]   Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 11.494                     0.30988              0.339
Sev   217.5    217.5 -1.246e-07  1.2656    1.2656   1.5532      1.5532
Agg    2500     2500 -1.246e-07 0.48516   0.48516  0.64857     0.64857
log2 = 16, bandwidth = 1/4, validation: not unreasonable.
```

We use the function `make_ceder_netter` to model the commission function. It takes a list of triples `(s, y, a)` as argument, interpreted as a share `s` of the layer `y` excess `a`. It returns two functions, a netter and a ceder, that map a subject loss to net or ceded. Multiple non-overlapping layers can be provided. They are combined into a single function. We will model the slide as the maximum 40% commission minus a cession to two layers with different shares. The required layer descriptions, in loss ratio points, are

- Layer 1 `(0.25, 0.2, 0.35)` for the slide between 35% and 55% and
- Layer 2 `(0.5, 0.1, 0.55)` for the slide between 55% and 65%.

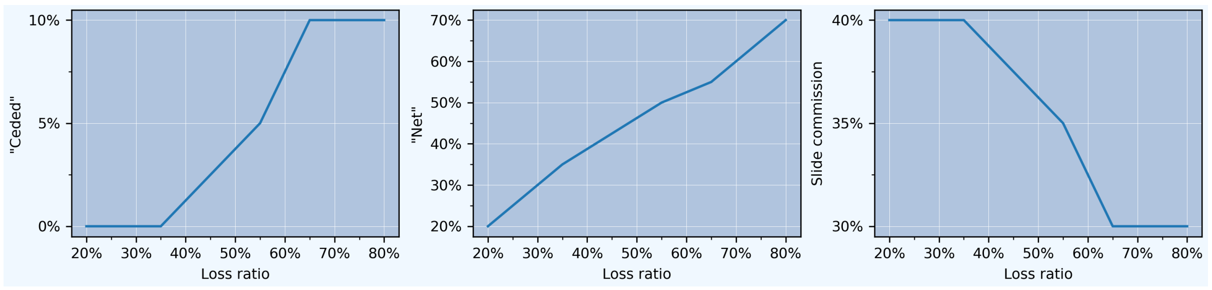
The function giving the slide payoff is easy to create, using a Python `lambda` function. The figure illustrates the ceder and netter functions and the function that computes the slide.

```
In [153]: from aggregate import make_ceder_netter

In [154]: import matplotlib.pyplot as plt

In [155]: from matplotlib import ticker
```

(continues on next page)

(continued from previous page)

```
In [156]: c, n = make_ceder_netter([(0.25, .2, .35), (0.5, .1, .55)])

In [157]: f = lambda x: 0.4 - c(x);                         \
   .....: lrs = np.linspace(0.2, 0.8, 61);                  \
   .....: slide = f(lrs);                                   \
   .....: fig, axs = plt.subplots(1,3,figsize=(3*3.5, 2.45), constrained_
↪layout=True); \
   .....: ax0, ax1, ax2 = axs.flat;                         \
   .....: ax0.plot(lrs, c(lrs));                            \
   .....: ax0.set(xlabel='Loss ratio', ylabel='"Ceded"'); \
   .....: ax1.plot(lrs, n(lrs));                            \
   .....: ax1.set(xlabel='Loss ratio', ylabel='"Net"');    \
   .....: ax2.plot(lrs, slide);
   .....:

In [158]: for ax in axs.flat:
   .....:     ax.xaxis.set_major_locator(ticker.MultipleLocator(0.1))
   .....:     ax.xaxis.set_minor_locator(ticker.MultipleLocator(0.05))
   .....:     ax.xaxis.set_major_formatter(ticker.StrMethodFormatter('{x:.0%}'))
   .....:     ax.yaxis.set_major_formatter(ticker.StrMethodFormatter('{x:.0%}'))
   .....:     if ax is ax1:
   .....:         ax.yaxis.set_major_locator(ticker.MultipleLocator(0.1))
   .....:         ax.yaxis.set_minor_locator(ticker.MultipleLocator(0.05))
   .....:     else:
   .....:         ax.yaxis.set_major_locator(ticker.MultipleLocator(0.05))
   .....:         ax.yaxis.set_minor_locator(ticker.MultipleLocator(0.025))
   .....:     ax.grid(lw=.25, c='w')
   .....:

In [159]: ax2.set(xlabel='Loss ratio', ylabel='Slide commission');
```



The expected commission across the estimated aggregate distribution can be computed by hand.

```
In [160]: subject = 25000;  re_rate = 0.2;  re_premium = subject * re_rate

In [161]: bit = a18.density_df[['loss', 'p_total']]

In [162]: bit['lr'] = bit.loss / re_premium

In [163]: bit['slide'] = f(bit.lr)

In [164]: (bit.slide * bit.p_total).sum()
Out[164]: 0.3574273911759826
```

The same quantity can be estimated using a lognormal approximation and numerical integration. The second value returned by `quad` estimates the relative error of the answer.

```
In [165]: import scipy.stats as ss

In [166]: mu, sigma = mu_sigma_from_mean_cv(0.5, 0.485)
```

(continues on next page)

(continued from previous page)

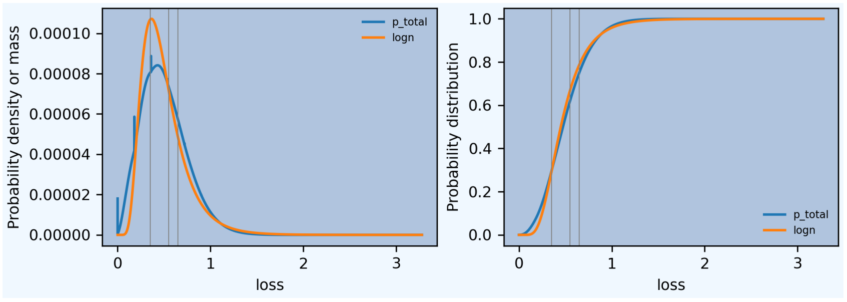
```
In [167]: fz = ss.lognorm(sigma, scale=np.exp(mu))

In [168]: quad(lambda x: (0.4 - c(x)) * fz.pdf(x), 0, np.inf)
Out[168]: (0.3613013481625082, 5.33305158633741e-09)
```

Bear and Nemlick use a coarser lognormal approximation to estimate the slide commission, Appendix K Exhibit I.

```
In [169]: mu, sigma = mu_sigma_from_mean_cv(1, 0.485)

In [170]: lr = 0.5; max_slide = 0.4

In [171]: entry_ratios = [1.3, 1.1, 0.7, 0]

In [172]: ins_charge = [1 - lognorm_lev(mu, sigma, 1, i) for i in entry_ratios]

In [173]: lr_points = np.diff(np.array(ins_charge), prepend=0) * lr

In [174]: slide_scale = np.array([0, .5, .75, 0])

In [175]: red_from_max = slide_scale * lr_points

In [176]: expected_slide = max_slide - np.sum(red_from_max)

In [177]: expected_slide
Out[177]: 0.3106485062930016
```

The lognormal distribution is not a great fit to the specified distribution.

```
In [178]: bit['logn'] = fz.pdf(bit.loss / re_premium)

In [179]: bit.logn = bit.logn / bit.logn.sum()

In [180]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True)

In [181]: ax0, ax1 = axs.flat

In [182]: bit.index = bit.index / re_premium

In [183]: bit[['p_total', 'logn']].plot(ax=ax0);

In [184]: bit[['p_total', 'logn']].cumsum().plot(ax=ax1);

In [185]: for ax in axs.flat:
   .....:     for lr in [.35, .55, .65]:
   .....:         ax.axvline(lr, lw=.5, c='C7')
   .....:

In [186]: ax0.set(ylabel='Probability density or mass');

In [187]: ax1.set(ylabel='Probability distribution');
```



TODO: investigate differences!

2.6.9 Outwards Analysis

Bear and Nemlick's analysis starts with a description of the frequency and severity of ceded loss. They do not consider the gross portfolio from which the cession occurs. In this section, we model gross, ceded, and net portfolios, mimicking a ceded re or broker actuary evaluation. We use an example from Mata *et al.* [2002]. Our methods are similar in spirit to theirs, but the details are slightly different, and our estimates do not tie exactly to what they report.

Subject business.

Lawyers and Errors and Omissions (E&O).

- Lawyers
  - Severity curve: lognormal $\mu = 8,\ \sigma = 2.5$
  - Loss ratio 65%
  - Exposure
    - 1M premium written with a 750K limit and 10K deductible
    - 2M premium written with a 1M limit and 25K deductible
- E&O
  - Severity curve: lognormal $\mu = 9,\ \sigma = 3$
  - Loss ratio 75%
  - Exposure
    - 2M premium written with a 1.5M limit and 50K deductible
    - 3M premium written with a 2M limit and 50K deductible

The total premium equals 8M, assumed split 7.2M for the first million and 800K for the second.

Cessions.

- Layer 1: 500 xs 500
  - Margin plus (retro) rated with provisional rate 12.5% of the premium for the first million, a minimum of 7%, maximum of 18%, and a load (lcf) of 107.5%.
  - Profit commission of 15% after 20% expenses.
  - Brokerage: 10% of provisional.
- Layer 2: 1M xs 1M
  - Cessions rated, 800K ceded premium
  - 15% ceding commission
  - Profit commission 15% after 20%
  - Brokerage: 10% on gross.

Treaty pricing with these variable features follows the same pattern as Bear and Nemlick and is left as an exercise. This section works with the gross, ceded, and net severity distributions, accounting for the limit profile, and the gross, ceded, and net aggregate distributions.

Stochastic Model

Mata et al. assume a negative binomial (gamma mixed Poisson) frequency distribution with a variance to mean ratio of 2.0. When there are relatively few excess claims the resulting mixing CV is close to 0 and the negative binomial is close to a Poisson. We start using a Poisson frequency and then show the impact of moving to a negative binomial.

The basic stochastic model is as follows. Work in 000s. Using `bs=1/2` results in a slightly better match to the mean and CV than the recommended `bs=1/4`.

```
In [188]: a19 = build('agg Re:MFV41 '
   .....:            '[1000 2000 2000 3000] premium at [.65 .65 .75 .75] lr '
   .....:            '[750 1000 1500 2000] xs [10 25 50 50] '
   .....:            'sev [exp(8)/1000 exp(8)/1000 exp(9)/1000 exp(9)/1000] '
   .....:            '* lognorm [2.5 2.5 3 3]  '
   .....:            'poisson', bs=1/2)
   .....:

In [189]: qd(a19)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  22.51                      0.21077            0.21077
Sev  253.23   253.23 -9.8245e-07  1.7344    1.7344   2.5007      2.5007
Agg    5700     5700 -9.8245e-07 0.42198   0.42198  0.60601     0.60601
log2 = 16, bandwidth = 1/2, validation: not unreasonable.
```

The `report_df` dataframe shows the theoretic and empirical (i.e., modeled) statistics for each unit.

```
In [190]: qd(a19.report_df.iloc[:, [0,1,2,3,4,-2]])

view                0         1         2         3 independent empirical
statistic
name          Re:MFV41  Re:MFV41  Re:MFV41  Re:MFV41    Re:MFV41
limit              750      1000      1500      2000      1202.5
attachment          10        25        50        50      29.468
el                 650      1300      1500      2250        5700
freq_m          6.4105    8.2296     3.416    4.4534       22.51
freq_cv        0.39496   0.34859   0.54105   0.47386     0.21077
freq_skew      0.39496   0.34859   0.54105   0.47386     0.21077
sev_m            101.4    157.97     439.1    505.23      253.23    253.23
sev_cv          1.7775    1.6113    1.2123    1.3296      1.7344    1.7344
sev_skew        2.5869    2.3057    1.1514    1.4005      2.5007    2.5007
agg_m              650      1300      1500      2250        5700      5700
agg_cv         0.80551   0.66106   0.85026   0.78834     0.42198   0.42198
agg_skew        1.1642   0.94224      1.04   0.98741     0.60601   0.60601
```

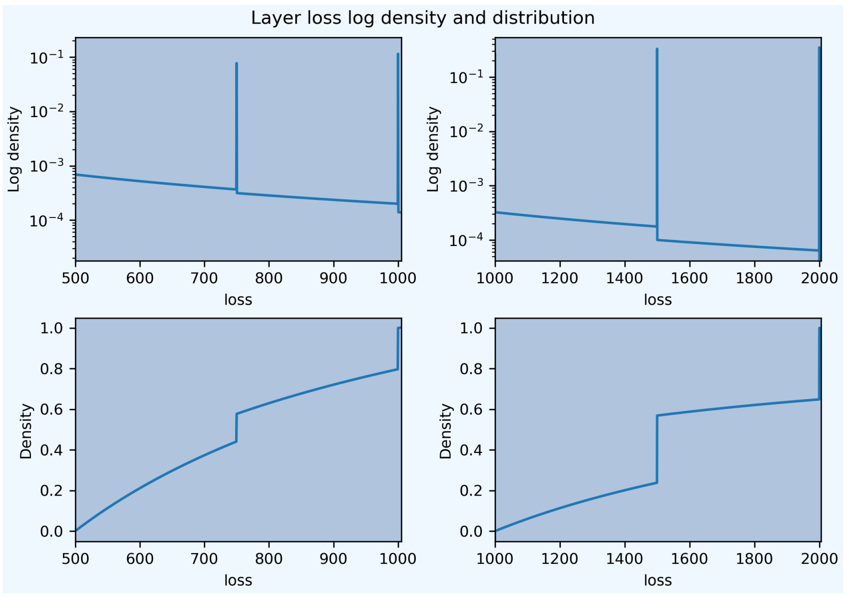
Mata et al. pay careful attention to the implied severity in each ceded layer, accounting for probability masses. They do this by considering losses in small intervals and weighting the underlying severity curves. `aggregate` automatically performs the same calculations to estimate the total layer severity. In this example, it uses a smaller bucket size of 0.5K compared to 2.5K in the original paper. The next plots reproduce [TODO Differences?!] Figures 2 and 3. The masses (spikes in density; jumps in distribution) occur when the lower limit unit has only limit losses.

```
In [191]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.45), constrained_
↪layout=True)

In [192]: ax0, ax1, ax2, ax3 = axs.flat

In [193]: (a19.density_df.p_sev / a19.sev.sf(500)).plot(xlim=[500, 1005],  ␣
↪logy=True, ax=ax0);

In [194]: (a19.density_df.p_sev / a19.sev.sf(1000)).plot(xlim=[1000, 2005],␣
↪logy=True, ax=ax1);
```

(continues on next page)

(continued from previous page)

```
In [195]: ((a19.density_df.F_sev - a19.sev.cdf(500)) / (a19.sev.cdf(1000) - a19.
↪sev.cdf(500))).plot(xlim=[500, 1005], ylim=[-0.05, 1.05], ax=ax2);

In [196]: ((a19.density_df.F_sev - a19.sev.cdf(1000)) / (a19.sev.cdf(2000) - a19.
↪sev.cdf(1000))).plot(xlim=[1000, 2005], ylim=[-0.05, 1.05], ax=ax3);

In [197]: for ax, y in zip(axs.flat, ['Log density', 'Log density', 'Density',
↪'Density']):
   .....:     ax.set(ylabel=y);
   .....:

In [198]: fig.suptitle('Layer loss log density and distribution');
```



Use an `occurrence net of` clause to apply the two excess of loss reinsurance layers. The estimated statistics refer to the net portfolio and reflect a pure exposure rating approach. Gross, ceded, and net expected losses are reported last.

```
In [199]: a19n = build('agg Re:MFV41n '
   .....:             '[1000 2000 2000 3000] premium at [.65 .65 .75 .75] lr '
   .....:             '[750 1000 1500 2000] xs [10 25 50 50] '
   .....:             'sev [exp(8)/1000 exp(8)/1000 exp(9)/1000 exp(9)/1000] *␣
↪lognorm [2.5 2.5 3 3]  '
   .....:             'occurrence net of 500 xs 500 and 1000 xs 1000 '
   .....:             'poisson', bs=1/2)
   .....:

In [200]: qd(a19n)

        E[X] Est E[X] Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  22.51                     0.21077            0.21077
Sev  253.23   153.57 -0.39356  1.7344    1.1741   2.5007      1.0678
Agg    5700   3456.7 -0.39356 0.42198   0.32507  0.60601     0.39439
log2 = 16, bandwidth = 1/2, validation: n/a, reinsurance.

In [201]: print(f'Gross expected loss {a19.est_m:,.1f}\n'
   .....:       f'Ceded expected loss {a19.est_m - a19n.est_m:,.1f}\n'
   .....:       f'Net expected loss   {a19n.est_m:,.1f}')
   .....:
```

(continues on next page)

(continued from previous page)

```
Gross expected loss 5,700.0
Ceded expected loss 2,243.3
Net expected loss   3,456.7
```

The `reinsurance_audit_df` dataframe summarizes ground-up (unconditional) layer loss statistics for occurrence covers. Thus, `ex` reports the layer severity per ground-up claim. The subject (gross) row is the same for all layers and replicates the gross severity statistics shown above for `a`.

```
In [202]: qd(a19n.reinsurance_audit_df.stack(0), sparsify=False)

                                     ex       var     sd     cv   skew
kind share limit  attach
occ  1.0   500.0  500.0  ceded   57.296     22247 149.15 2.6032  2.427
occ  1.0   500.0  500.0  net     195.93     93447 305.69 1.5602 2.5723
occ  1.0   500.0  500.0  subject 253.23 1.929e+05 439.21 1.7344 2.5007
occ  1.0   1000.0 1000.0 ceded   42.365     31584 177.72  4.195 4.4786
occ  1.0   1000.0 1000.0 net     210.86     94455 307.34 1.4575 1.6901
occ  1.0   1000.0 1000.0 subject 253.23 1.929e+05 439.21 1.7344 2.5007
occ  all   inf    gup    ceded    99.66     91341 302.23 3.0326 3.4354
occ  all   inf    gup    net     153.57     32509  180.3 1.1741 1.0678
occ  all   inf    gup    subject 253.23 1.929e+05 439.21 1.7344 2.5007
```

The `reinsurance_occ_layer_df` dataframe summarizes aggregate losses.

```
In [203]: qd(a19n.reinsurance_occ_layer_df, sparsify=False)

stat                    ex     ex      ex     cv     cv      cv     en severity  ␣
↪ pct
view                 ceded    net subject  ceded    net subject  ceded    ceded  ␣
↪ceded
share limit  attach                                                              ␣
↪
1.0   500.0  500.0  1289.7 4410.3    5700 2.6032 1.5602  1.7344 3.5198   366.41 0.
↪22626
1.0   1000.0 1000.0 953.61 4746.4    5700  4.195 1.4575  1.7344 1.5175   628.41  0.
↪1673
all   inf    gup    2243.3 3456.7    5700 3.0326 1.1741  1.7344  22.51    99.66 0.
↪39356
```

The layer severities show above differ slightly from Mata et al. Table 3. The `aggregate` computation is closest to Method 3. The reported severities are 351.1 and 628.8.

The `reinsurance_df` dataframe provides the gross, ceded, and net severity and aggregate distributions:

- Severity distributions: `p_sev_gross`, `p_sev_ceded`, `p_sev_net`
- Aggregate distribution: `p_agg_gross_occ`, `p_agg_ceded_occ`, `p_agg_net_occ` show the aggregate distributions computed using gross, cede, and net severity (occurrence) distributions. These are the portfolio gross, ceded and net distributions.
- The columns `p_agg_gross`, `p_agg_ceded`, `p_agg_net` are relevant only when there is are `occurrence` and `aggregate` reinsurance clauses. They report gross, ceded and net of the aggregate covers, using the severity requested in the occurrence clause. In this case `p_agg_gross` is the same as `p_agg_net_occ` because the occurrence clause specified `net of`.

Here is an extract from the severity distributions. Ceded severity is at most 1500. The masses at 250, 500, 1000 and 1500 are evident.

```
In [204]: qd(a19n.reinsurance_df.loc[0:2000:250,
   .....:      ['p_sev_gross', 'p_sev_ceded', 'p_sev_net']])
   .....:
```

(continues on next page)

(continued from previous page)

```
        p_sev_gross  p_sev_ceded  p_sev_net
loss
0.0       0.0059219      0.84369  0.0059219
125.0    0.00074474   7.6472e-05 0.00074474
250.0    0.00029874     0.012139 0.00029874
375.0    0.00016663   3.8634e-05 0.00016663
500.0    0.00010806     0.018065    0.15642
625.0    7.6472e-05   1.8367e-05          0
750.0      0.012139   1.5689e-05          0
875.0    3.8634e-05   1.3584e-05          0
1000.0     0.018065     0.022295          0
1125.0   1.8367e-05   5.9515e-06          0
1250.0   1.5689e-05   5.3056e-06          0
1375.0   1.3584e-05   4.7639e-06          0
1500.0     0.022295     0.023679          0
1625.0   5.9515e-06            0          0
1750.0   5.3056e-06            0          0
1875.0   4.7639e-06            0          0
2000.0     0.023679            0          0
```

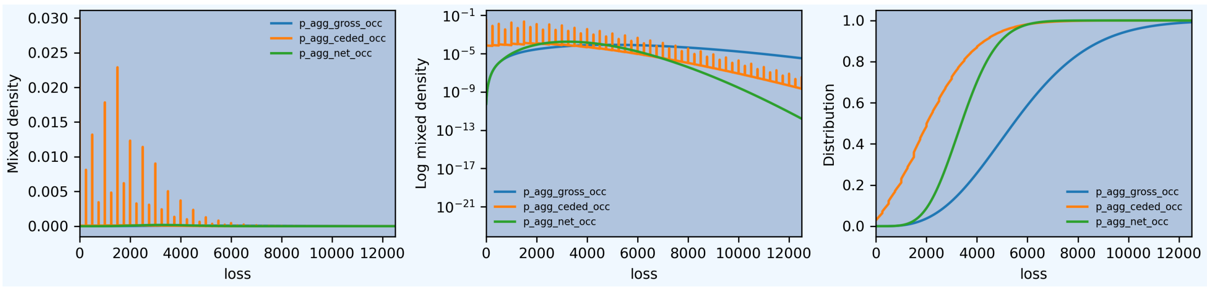
Here is an extract from the aggregate distributions, followed by the density and distribution plots. The masses are caused by outcomes involving only limit losses.

```
In [205]: qd(a19n.reinsurance_df.loc[3000:6000:500,
   .....:     ['p_agg_gross_occ', 'p_agg_ceded_occ', 'p_agg_net_occ']])
   .....:

        p_agg_gross_occ  p_agg_ceded_occ  p_agg_net_occ
loss
3000.0       5.6326e-05        0.0090335     0.00017503
3250.0       6.2846e-05        0.0024374     0.00017886
3500.0       6.8727e-05        0.0050447     0.00017401
3750.0       7.3779e-05        0.0013535     0.00016174
4000.0       7.7914e-05        0.0037091     0.00014418
4250.0       8.1081e-05        0.0010054     0.00012359
4500.0        8.328e-05         0.002359     0.00010216
4750.0       8.4511e-05       0.00063735     8.1596e-05
5000.0       8.4787e-05        0.0012892     6.3111e-05
5250.0       8.4135e-05       0.00034824      4.735e-05
5500.0       8.2634e-05       0.00080992     3.4517e-05
5750.0       8.0373e-05       0.00022017     2.4482e-05
6000.0       7.7463e-05       0.00045425     1.6918e-05

In [206]: fig, axs = plt.subplots(1, 3, figsize=(3 * 3.5, 2.45), constrained_
↪layout=True)

In [207]: ax0, ax1, ax2 = axs.flat

In [208]: bit = a19n.reinsurance_df[['p_agg_gross_occ', 'p_agg_ceded_occ', 'p_agg_
↪net_occ']]

In [209]: bit.plot(ax=ax0);

In [210]: bit.plot(logy=True, ax=ax1);

In [211]: bit.cumsum().plot(ax=ax2);

In [212]: for ax in axs.flat:
   .....:     ax.set(xlim=[0, 12500]);
   .....:
```

(continues on next page)

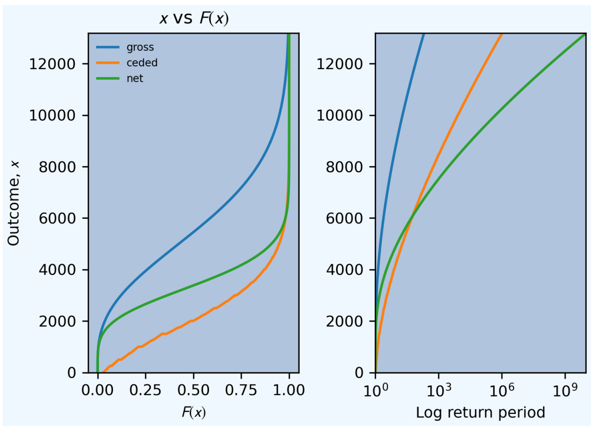
(continued from previous page)

```
In [213]: ax0.set(ylabel='Mixed density');

In [214]: ax1.set(ylabel='Log mixed density');

In [215]: ax2.set(ylabel='Distribution');
```



Any desired risk management evaluation can be computed from `reinsurance_df`, which contains the gross, ceded, and net distributions. For example, here is a tail return period plot and a dataframe of summary statistics.

```
In [216]: fig, axs = plt.subplots(1, 2, figsize=(2 * 2.45, 3.5), constrained_
↪layout=True)

In [217]: ax0, ax1 = axs.flat

In [218]: for c in bit.columns:
   .....:     ax0.plot(bit[c].cumsum(), bit.index, label=c.split('_')[2])
   .....:     rp = 1 / (1 - bit[c].cumsum())
   .....:     ax1.plot(rp, bit.index, label=c)
   .....:

In [219]: ax0.xaxis.set_major_locator(ticker.MultipleLocator(0.25))

In [220]: ax0.set(ylim=[0, a19n.q(1-1e-10)], title='$x$ vs $F(x)$', xlabel='$F(x)$
↪', ylabel='Outcome, $x$');

In [221]: ax1.set(xscale='log', xlim=[1, 1e10], ylim=[0, a19n.q(1-1e-10)], xlabel=
↪'Log return period');

In [222]: ax0.legend(loc='upper left');

In [223]: df = pd.DataFrame({c.split('_')[2]: xsden_to_meancvskew(bit.index,␣
↪bit[c]) for c in bit.columns},
   .....:                 index=['mean', 'cv', 'skew'])
   .....:

In [224]: qd(df)

       gross   ceded     net
mean    5700  2243.3  3456.7
cv   0.42198 0.67304 0.32507
skew 0.60601  0.8053 0.39439
```



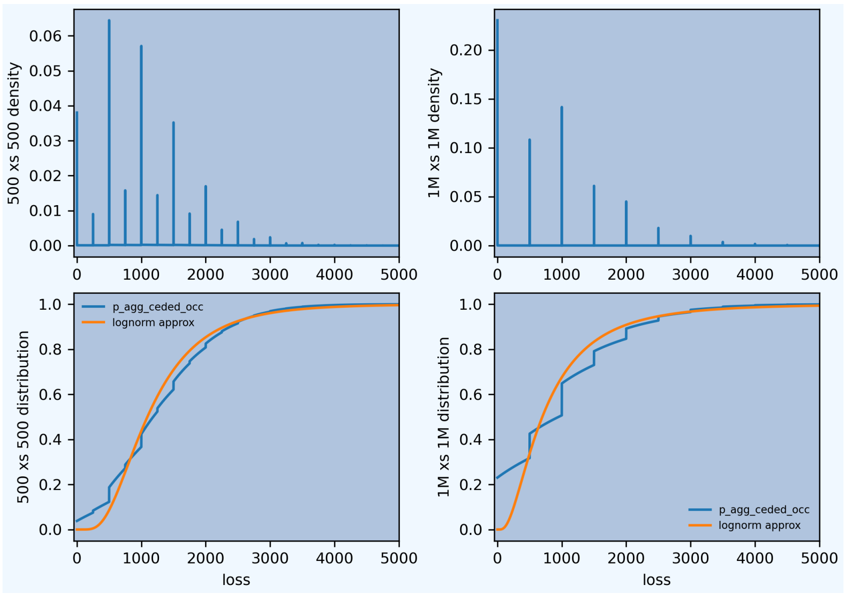
Mata Figures 4, 5, 6 and 7 show the aggregate mixed density and distribution functions for each layer. These plots are replicated below. Our model uses a gamma mixed Poisson frequency with a variance multiplier of 2.0, resulting in a lower variance multiplier for excess layers (see REF). The plots in Mata appear to use a variance multiplier of 2.0 for the excess layer, resulting in a more skewed distribution.

```
In [225]: from aggregate import lognorm_approx

In [226]: vm = 2.0; c = (vm - 1) / a19.n; cv = c**0.5

In [227]: a20 = build('agg Re:MFV41n1 '
   .....:             '[1000 2000 2000 3000] premium at [.65 .65 .75 .75] lr '
   .....:             '[750 1000 1500 2000] xs [10 25 50 50] '
   .....:             'sev [exp(8)/1000 exp(8)/1000 exp(9)/1000 exp(9)/1000] *␣
↪lognorm [2.5 2.5 3 3]  '
   .....:             'occurrence net of 500 xs 500 '
   .....:             f'mixed gamma {cv}', bs=1/2)
   .....:

In [228]: a21 = build('agg Re:MFV41n2 '
   .....:             '[1000 2000 2000 3000] premium at [.65 .65 .75 .75] lr '
   .....:             '[750 1000 1500 2000] xs [10 25 50 50] '
   .....:             'sev [exp(8)/1000 exp(8)/1000 exp(9)/1000 exp(9)/1000] *␣
↪lognorm [2.5 2.5 3 3]  '
   .....:             'occurrence net of 1000 xs 1000 '
   .....:             f'mixed gamma {cv}', bs=1/2)
   .....:

In [229]: qd(a20)

       E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  22.51                   0.29808            0.44712
Sev  253.23   195.93 -0.22626  1.7344    1.5602   2.5007      2.5723
Agg    5700   4410.3 -0.22626  0.4717   0.44384  0.69763     0.68565
log2 = 16, bandwidth = 1/2, validation: n/a, reinsurance.

In [230]: qd(a21)

       E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  22.51                   0.29808            0.44712
Sev  253.23   210.86  -0.1673  1.7344    1.4575   2.5007      1.6901
Agg    5700   4746.4  -0.1673  0.4717   0.42805  0.69763     0.60342
log2 = 16, bandwidth = 1/2, validation: n/a, reinsurance.

In [231]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.45), constrained_
↪layout=True); \
```

(continues on next page)

(continued from previous page)

```
   .....: ax0, ax1, ax2, ax3 = axs.flat; \
   .....: a20.reinsurance_df.p_agg_ceded_occ.plot(ax=ax0); \
   .....: a20.reinsurance_df.p_agg_ceded_occ.cumsum().plot(ax=ax2); \
   .....: a21.reinsurance_df.p_agg_ceded_occ.plot(ax=ax1); \
   .....: a21.reinsurance_df.p_agg_ceded_occ.cumsum().plot(ax=ax3); \
   .....: xs = np.linspace(0, 5000, 501); \
   .....: fz = lognorm_approx(a20.reinsurance_df.p_agg_ceded_occ); \
   .....: ax2.plot(xs, fz.cdf(xs), label='lognorm approx'); \
   .....: fz = lognorm_approx(a21.reinsurance_df.p_agg_ceded_occ); \
   .....: ax3.plot(xs, fz.cdf(xs), label='lognorm approx'); \
   .....: ax2.legend(); \
   .....: ax3.legend(); \
   .....: ax0.set(xlim=[-50, 5000], xlabel=None, ylabel='500 xs 500 density'); \
   .....: ax2.set(xlim=[-50, 5000], ylabel='500 xs 500 distribution'); \
   .....: ax1.set(xlim=[-50, 5000], xlabel=None, ylabel='1M xs 1M density');
   .....:

In [232]: ax3.set(xlim=[-50, 5000], ylabel='1M xs 1M distribution');
```



2.6.10 Summary of Objects Created by DecL

Objects created by `build()` in this guide.

```
In [233]: from aggregate import pprint_ex

In [234]: for n, r in build.qlist('^(Re):').iterrows():
   .....:     pprint_ex(r.program, split=20)
   .....:
```

2.7 Reserving

Objectives: Applications of the `Aggregate` class to reserving, including models of loss emergence and determining ranges for IBNR and case reserves.

Audience: Reserving, capital modeling, ERM actuaries.

Prerequisites: DecL, the reserving process and terminology, aggregate distributions, risk measures.

See also:

Contents:

2.7.1 Helpful References

- Meyers [2019]
- Mildenhall and Major [2022], chapter 17

2.7.2 Modeling the Current Accident Year, Case and IBRN Reserves

Todo: Documentation to follow.

2.7.3 The Resolution of Reserve Uncertainty Over Time

Todo: Documentation to follow.

2.8 Catastrophe Modeling

Objectives: Applications of the `Aggregate` class to catastrophe risk evaluation and pricing using thick-tailed Poisson Pareto and lognormal models, including occurrence and aggregate PMLs (OEP, AEP) and layer loss costs. Covers material on CAS Parts 8 and 9.

Audience: Catastrophe modelers, reinsurance actuaries, and risk management professionals.

Prerequisites: Basics of catastrophe modeling, catastrophe insurance and reinsurance terminology, use of `build`.

See also: *Capital Modeling and Risk Management*, *Strategy and Portfolio Management*, *Reinsurance Pricing*, *Individual Risk Pricing*.

Contents:

6. *Summary of Objects Created by DecL*

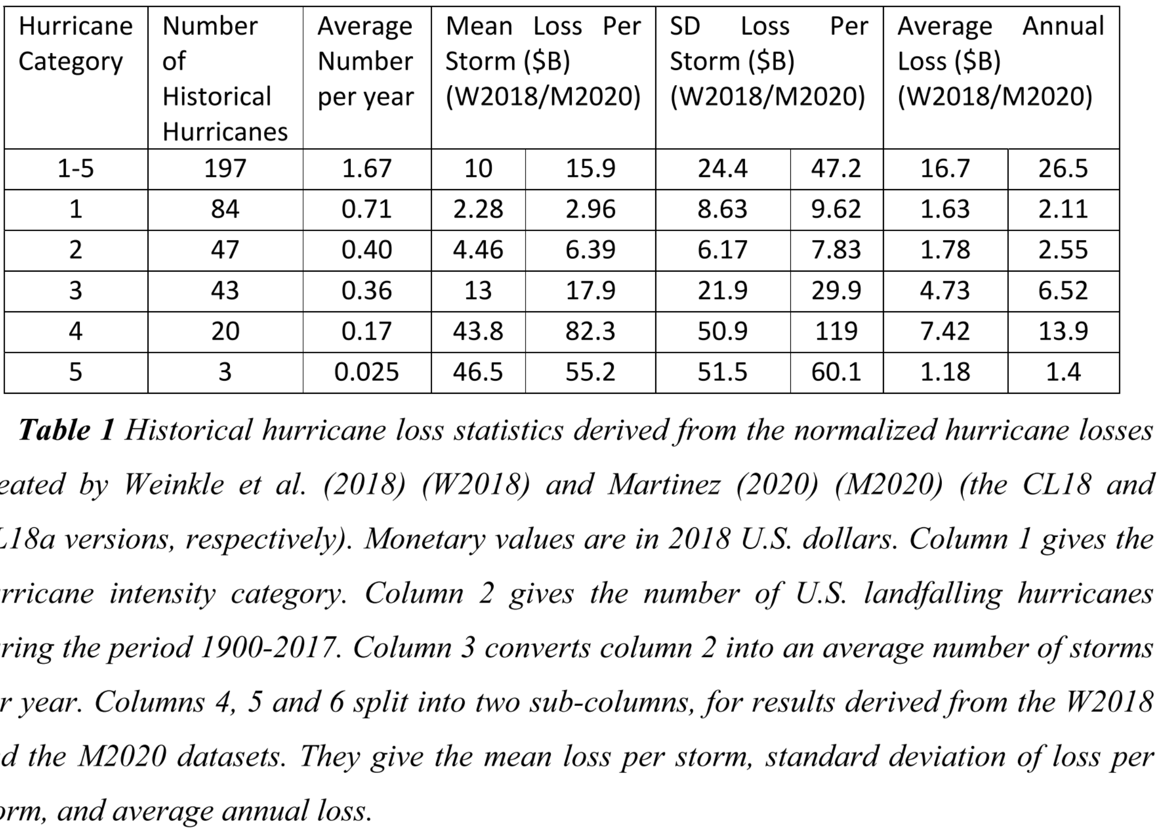
2.8.1 Helpful References

- Jewson [2022]
- Mitchell-Wallace *et al.* [2017]
- Anderson and Dong [1988]
- Woo [2002]

2.8.2 Jewson's US Wind PML Estimates

Model Description

Stephen Jewson *Projections of Changes in U.S. Hurricane Damage Due to Projected Changes in Hurricane Frequencies* (submitted, under peer review), Jewson [2022] reports the following frequency and severity statistics for US hurricane losses.



Hurricane Category	Number of Historical Hurricanes	Average Number per year	Mean Loss Per Storm ($B) (W2018/M2020)		SD Loss Per Storm ($B) (W2018/M2020)		Average Annual Loss ($B) (W2018/M2020)	
1-5	197	1.67	10	15.9	24.4	47.2	16.7	26.5
1	84	0.71	2.28	2.96	8.63	9.62	1.63	2.11
2	47	0.40	4.46	6.39	6.17	7.83	1.78	2.55
3	43	0.36	13	17.9	21.9	29.9	4.73	6.52
4	20	0.17	43.8	82.3	50.9	119	7.42	13.9
5	3	0.025	46.5	55.2	51.5	60.1	1.18	1.4

***Table 1** Historical hurricane loss statistics derived from the normalized hurricane losses created by Weinkle et al. (2018) (W2018) and Martinez (2020) (M2020) (the CL18 and CL18a versions, respectively). Monetary values are in 2018 U.S. dollars. Column 1 gives the hurricane intensity category. Column 2 gives the number of U.S. landfalling hurricanes during the period 1900-2017. Column 3 converts column 2 into an average number of storms per year. Columns 4, 5 and 6 split into two sub-columns, for results derived from the W2018 and the M2020 datasets. They give the mean loss per storm, standard deviation of loss per storm, and average annual loss.*

The dataframe `jewson` recreates the table and adds severity CVs.

```
In [1]: from aggregate import build, qd, mv

In [2]: import pandas as pd

In [3]: import matplotlib.pyplot as plt

In [4]: jewson = pd.DataFrame(
   ...:     {'Num': [197, 84, 47, 43, 20, 3],
   ...:     'EN': [1.67, 0.71, 0.4, 0.36, 0.17, 0.025],
```

(continues on next page)

(continued from previous page)

```
   ...:     'ES_W': [10.0, 2.28, 4.46, 13.0, 43.8, 46.5],
   ...:     'ES_M': [15.9, 2.96, 6.39, 17.9, 82.3, 55.2],
   ...:     'SD_W': [24.4, 8.63, 6.17, 21.9, 50.9, 51.5],
   ...:     'SD_M': [47.2, 9.62, 7.83, 29.9, 119.0, 60.1],
   ...:     'EX_W': [16.7, 1.63, 1.78, 4.73, 7.42, 1.18],
   ...:     'EX_M': [26.5, 2.11, 2.55, 6.52, 13.9, 1.4]},
   ...:     index=pd.Index(['1-5', '1', '2', '3', '4', '5'],
   ...:     dtype='object', name='Cat')
   ...:     )
   ...:

In [5]: jewson['CV_W'] = jewson.SD_W / jewson.ES_W;   \
   ...: jewson['CV_M'] = jewson.SD_M / jewson.ES_M
   ...:

In [6]: qd(jewson)

     Num    EN  ES_W  ES_M  SD_W  SD_M  EX_W  EX_M   CV_W   CV_M
Cat
1-5  197  1.67    10  15.9  24.4  47.2  16.7  26.5   2.44 2.9686
1     84  0.71  2.28  2.96  8.63  9.62  1.63  2.11 3.7851   3.25
2     47   0.4  4.46  6.39  6.17  7.83  1.78  2.55 1.3834 1.2254
3     43  0.36    13  17.9  21.9  29.9  4.73  6.52 1.6846 1.6704
4     20  0.17  43.8  82.3  50.9   119  7.42  13.9 1.1621 1.4459
5      3 0.025  46.5  55.2  51.5  60.1  1.18   1.4 1.1075 1.0888
```

Jewson models aggregate losses with Poisson frequency and lognormal severity assumptions. Use `build` to create `Aggregate` models of the two implied distributions. Adjust `bs` from recommended 1/16 to 1/8 for thick tailed distributions.

```
In [7]: w = build('agg Cat:USWind:W '
   ...:           f'{jewson.loc["1":"5", "EN"].to_numpy()} claims '
   ...:           f'sev lognorm {jewson.loc["1":"5", "ES_W"].to_numpy()} '
   ...:           f'cv {jewson.loc["1":"5", "CV_W"].to_numpy()}'
   ...:           'poisson'
   ...:           , bs=1/8)
   ...:

In [8]: m = build('agg Cat:USWind:M: '
   ...:           f'{jewson.loc["1":"5", "EN"].to_numpy()} claims '
   ...:           f'sev lognorm {jewson.loc["1":"5", "ES_M"].to_numpy()} '
   ...:           f'cv {jewson.loc["1":"5", "CV_M"].to_numpy()}'
   ...:           'poisson'
   ...:            , bs=1/8 )
   ...:

In [9]: qd(w)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  1.665                      0.77498            0.77498
Sev  10.025   10.024 -4.5361e-05  2.4845    2.4841   9.9667      9.7215
Agg  16.691   16.691 -4.5361e-05  2.0756    2.0753   6.9539      6.8019
log2 = 16, bandwidth = 1/8, validation: fails sev skew, agg skew.

In [10]: mv(w)
mean     = 16.6913
variance = 1200.188
std dev  = 34.6437

In [11]: qd(m)
```

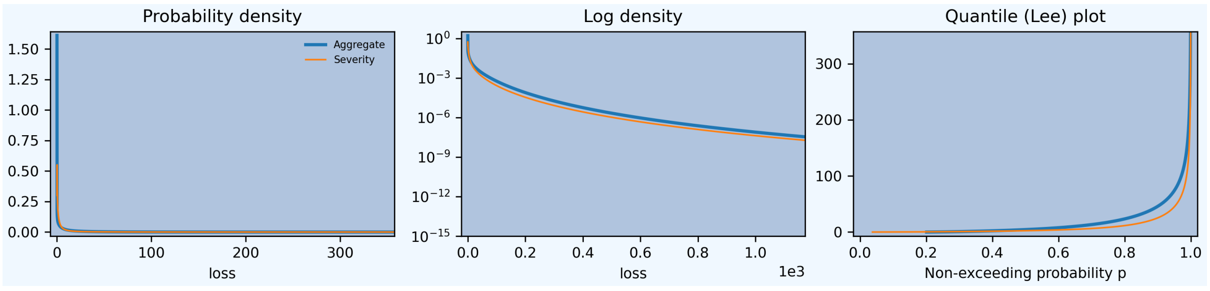
(continues on next page)

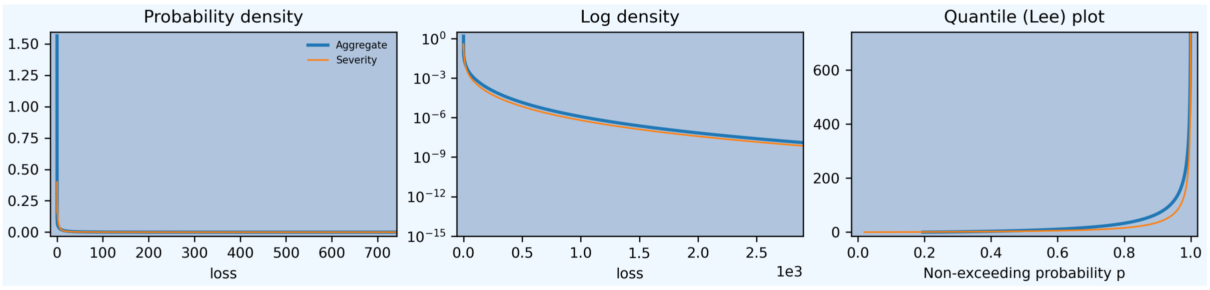
(continued from previous page)

```
       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq  1.665                      0.77498            0.77498
Sev  15.899   15.899 -4.9864e-05  3.0261    3.0218   15.065      14.399
Agg  26.473   26.471 -4.9871e-05  2.4699    2.4668   10.676       10.23
log2 = 16, bandwidth = 1/8, validation: fails sev cv, agg cv.

In [12]: mv(m)
mean     = 26.4726
variance = 4275.28
std dev  = 65.3856
```

Plots of the severity and aggregate distributions confirms they are very thick tailed.

```
In [13]: w.plot()

In [14]: m.plot()
```





Aggregate PML Estimates

It is easy to compute aggregate PML points (aggregate quantiles). The next table shows values at a range of return periods. The return period corresponding to a p quantile is $1/(1-p)$. In a Poisson frequency model, the reciprocal of the frequency equals the average waiting time between events because of the relationship between the Poisson and exponential distributions. Amounts are in USD billions.

```
In [15]: agg_pmls = pd.DataFrame({'Return': [2, 5, 10, 20, 25, 50, 100, 200, 250,␣
↪1000, 10000]}, dtype=float)

In [16]: agg_pmls['p'] = 1 - 1/agg_pmls.Return

In [17]: agg_pmls['Weinkle'] = [w.q(i) for i in agg_pmls.p]

In [18]: agg_pmls['Martinez'] = [m.q(i) for i in agg_pmls.p]

In [19]: agg_pmls = agg_pmls.set_index(['Return'])

In [20]: qd(agg_pmls)
```

(continues on next page)

(continued from previous page)

```
               p  Weinkle  Martinez
Return
2.0          0.5      4.5     6.375
5.0          0.8     23.5        33
10.0         0.9       46      68.5
20.0        0.95   73.875    117.38
25.0        0.96   84.125    136.12
50.0        0.98   119.25    204.12
100.0       0.99   160.38    288.88
200.0      0.995   208.12    392.88
250.0      0.996   225.12       431
1000.0     0.999    350.5       727
10000.0 0.9999   657.88      1516
```

Occurrence PML Estimates

Occurrence PMLs, called **occurrence exceeding probability** (OEP) points, can be computed for a compound Poisson model as adjusted severity quantiles. The n year OEP is defined as the loss level $\mathit{OEP}(n)$ so that

> there is a $1/n$ chance of one or more losses greater than $\mathit{OEP}(n)$ per year.

The definition is valid only for $n \ge 1$ since $1/n$ is interpreted as a probability. If λ is the annual event frequency and S the severity survival function, then the annual frequency of losses greater than x equals $\lambda S(x)$ and therefore chance of one or more losses greater than x equals $1 - \exp(-\lambda S(x))$ with Poisson frequency (one minus chance of no events). Rearranging gives

$$\mathit{OEP}(n) = q\left(1 + \frac{\log(1 - 1/n)}{\lambda}\right)$$

where q is the severity distribution quantile function.

Jewson [2022] considers the related notion of **event exceedance frequency** (EEF). Here, the n year EEF is defined as the loss level $\mathit{EEF}(n)$ with a $1/n$ annual frequency. Thus

$$\mathit{EEF}(n) = q\left(1 - \frac{1}{\lambda n}\right).$$

This definition is valid for any $n > 0$ since the result is a frequency rather than a probability. OEP and EEF are very similar for large n because $\log(1 + x) \approx x$ for small x, but they diverge significantly for small n and only the latter makes sense for $0 < n < 1$. Jewson shows EEFs in his Figure 2. See Aggregate and Occurrence Probable Maximal Loss and Catastrophe Model Output.

Jewson comments that OEP is useful for validating the frequency of annual maximum loss, which is affected by clustering. Thus OEP estimates are important for validating models that include clustering. EEF is useful for validating whether a model captures the mean frequency of events, including events with frequency greater than 1 per year.

The following table shows OEP and EEF points, comparing the two statistics.

```
In [21]: oep = pd.DataFrame({'Return': [2, 5, 10, 20, 25, 50, 100, 200, 250, 1000,␣
↪10000]}, dtype=float); \
   ....: oep['p'] = 1 - 1/oep.Return;                                   \
   ....: oep['W OEP'] = [w.q_sev(1 + np.log(i) / w.n) for i in oep.p];      \
   ....: oep["W EEF"] = [w.q_sev(1 - 1 / i /w.n) for i in oep.Return];     \
   ....: oep['M OEP'] = [m.q_sev(1 + np.log(i) / m.n) for i in oep.p];      \
   ....: oep["M EEF"] = [m.q_sev(1 - 1 / i /m.n) for i in oep.Return];     \
   ....: oep = oep.set_index(['Return']);                               \
   ....: qd(oep)
   ....:

               p  W OEP  W EEF  M OEP  M EEF
Return
```

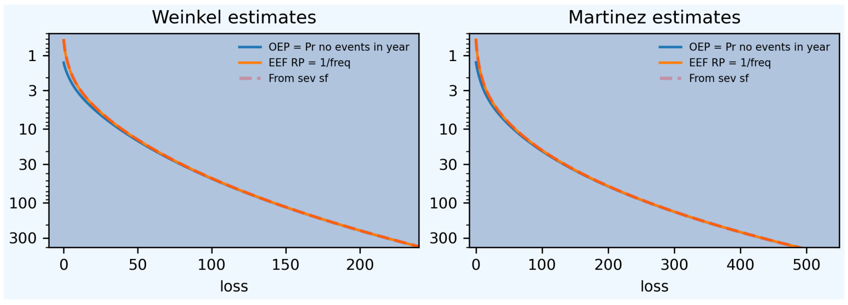
(continues on next page)

(continued from previous page)

```
2.0          0.5  3.625  6.375  5.125  8.875
5.0          0.8  18.75 21.125 26.125  29.75
10.0         0.9  37.75 39.375  56.75 59.375
20.0        0.95 62.125 63.125  100.5 102.38
25.0        0.96  71.25     72 117.62 119.25
50.0        0.98 103.12 103.62    181    182
100.0       0.99 141.62 141.88 261.75 262.38
200.0      0.995 187.38 187.62 362.75 363.12
250.0      0.996 203.88    204 400.12 400.38
1000.0     0.999 327.38  327.5 693.38  693.5
10000.0 0.9999 635.38 635.38 1482.8 1482.8
```

The next block of code shows the same information as Jewson's Figure 2. It includes OEP and EEF for comparison. The dashed line shows a third alternative `aggregate` implementation using the exact continuous weighted severity survival function `m.sev.sf`, rather than the discrete approximation in `m.density_df`.

```
In [22]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True)

In [23]: for ax, mw, title, xmax in zip(axs.flat[::-1], [m, w], ['Martinez␣
↪estimates', 'Weinkel estimates'], [550, 240]):
   ....:     bit = np.exp(-(1-mw.density_df.F_sev.loc[:1000]) * m.n)
   ....:     bit = 1 / (1 - bit)
   ....:     bit.plot(logy=True, ax=ax, label='OEP = Pr no events in year')
   ....:     bit = 1 / ((1 - mw.density_df.F_sev.loc[:20000]) * m.n)
   ....:     bit.plot(logy=True, ax=ax, label='EEF RP = 1/freq')
   ....:     xs = np.linspace(0, 500, 501)
   ....:     if title[0] == 'M':
   ....:         rp = 1 / (m.n * m.sev.sf(xs))
   ....:     else:
   ....:         rp = 1 / (m.n * w.sev.sf(xs))
   ....:     ax.plot(xs, rp, c='r', ls='--', lw=2, alpha=0.25, label='From sev sf')
   ....:     ax.legend()
   ....:     ax.set(xlim=[-10, xmax], ylim=[400, .5], title=title)
   ....:     ax.set_yscale('log', base=10)
   ....:     ticks = [1, 3, 10, 30, 100, 300]
   ....:     ax.set_yticks(ticks)
   ....:     ax.set_yticklabels([f'{x}' for x in ticks]);
   ....:

In [24]: fig;
```



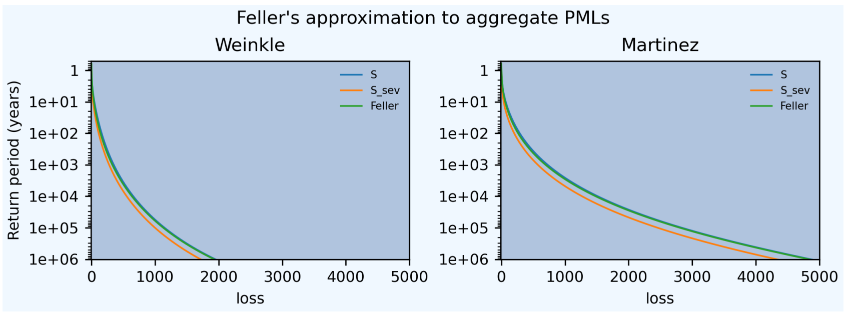
Feller's Relationship between AEP and OEP

For thick tailed distributions, AEP and OEP points are closely related by Feller's theorem, which says that for $A \sim \mathsf{CP}(\lambda, X)$ with severity X subexponential,

$$\lambda \Pr(X > x) \to \Pr(A > x)$$

as $x \to \infty$, see REF. The next plot confirms that Feller's approximation is very good. Note the extreme return periods returned by `aggregate` that would be hard to estimate with simulation.

```
In [25]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.55), constrained_
↪layout=True)

In [26]: for ax, mw, lim, title in zip(axs.flat[::-1], [m, w], [5000, 5000], [
↪'Martinez', 'Weinkle']):
   ....:     bit = mw.density_df.loc[:5000, ['S', 'S_sev']]
   ....:     bit['Feller'] = bit.S_sev * mw.n
   ....:     bit = 1 / bit
   ....:     bit.plot(xlim=[-10, lim], logy=True, ax=ax, ylim=[1000000, 0.5], lw=1)
   ....:     ax.set_yscale('log', base=10)
   ....:     ticks = [1,10,100,1000,10000, 1e5, 1e6]
   ....:     ax.set_yticks(ticks)
   ....:     ax.set_yticklabels([f'{x:.0g}' for x in ticks]);
   ....:     ax.set(title=title);
   ....:     if ax is axs[0]: ax.set(ylabel='Return period (years)');
   ....:

In [27]: fig.suptitle("Feller's approximation to aggregate PMLs")
Out[27]: Text(0.5, 0.98, "Feller's approximation to aggregate PMLs")
```



Note: These graphs demonstrate computational facility. I'm not suggesting one million year PML is a reliable estimate. But the figure is **reliably computing what the specified statistical model implies**. The losses shown range up to USD 5 trillion, about 20% of GDP.

2.8.3 Jewson's US Wind Climate Change Estimates

Jewson Table 2 provides estimates for the impact of a 2 degree Celcius increase in global mean surface temperature (GMST) on event frequency by Safir-Simpson category. He also provides the standard deviation of the impact. These are added in the next dataframe.

```
In [28]: jewson['Freq Chg'] = [None, 1.011, 1.095, 1.134, 1.179, 1.236]

In [29]: jewson['Freq Chg SD'] = [None, 0.3179, .4176, .4638, .5174, .5830]

In [30]: qd(jewson.loc["1":"5", ['Freq Chg', 'Freq Chg SD']])

     Freq Chg  Freq Chg SD
```

(continues on next page)

(continued from previous page)

```
Cat
1        1.011        0.3179
2        1.095        0.4176
3        1.134        0.4638
4        1.179        0.5174
5        1.236         0.583
```

He models the impact of climate change on PMLs by assuming the frequency of each category is perturbed using a lognormal with mean and standard deviation given by the last two columns of the above table. He assumes that the perturbations across categories are comonotonic. In actuarial terms, he is using comonotonic frequency mixing variables, to create a mixed compound Poisson.

We can create a similar effect using `aggregate` first by adjusting the baseline event frequencies by the `Freq Chg` column and then by applying shared mixing across all events together (resulting in comonotonic perturbations). We select a mix CV equal to Jewson's estimate for Category 4 events. The categories are similar — in light of the overall uncertainty of the analysis.

```
In [31]: qd((jewson.iloc[1:, -1] / jewson.iloc[1:, -2]))

Cat
1   0.31444
2   0.38137
3   0.40899
4   0.43885
5   0.47168

In [32]: mix_cv = 0.5174 / 1.179

In [33]: mix_cv
Out[33]: 0.4388464800678541
```

The adjusted model is built using inverse Gaussian mixing variables (slightly thicker tail than gamma), rather than Jewson's lognormals. Note that the standard deviations increase but the CVs decrease.

```
In [34]: wcc = build('agg Cat:USWind:Wcc '
   ....:             f'{jewson.loc["1":"5", "EN"].to_numpy() * jewson.loc["1":"5",
↪"Freq Chg"].to_numpy()} claims '
   ....:             f'sev lognorm {jewson.loc["1":"5", "ES_W"].to_numpy()} '
   ....:             f'cv {jewson.loc["1":"5", "CV_W"].to_numpy()}'
   ....:             f'mixed ig {mix_cv}'
   ....:             , bs=1/8)
   ....:

In [35]: mcc = build('agg Cat:USWind:Mcc '
   ....:             f'{jewson.loc["1":"5", "EN"].to_numpy() * jewson.loc["1":"5",
↪"Freq Chg"].to_numpy()} claims '
   ....:             f'sev lognorm {jewson.loc["1":"5", "ES_M"].to_numpy()} '
   ....:             f'cv {jewson.loc["1":"5", "CV_M"].to_numpy()}'
   ....:             f'mixed ig {mix_cv}'
   ....:              , bs=1/8 )
   ....:

In [36]: qd(wcc)

       E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq 1.7954                      0.86578              1.1454
Sev  10.646    10.645 -4.0062e-05  2.4249     2.4246    9.5485       9.3409
Agg  19.113    19.112 -4.0062e-05  2.0062      2.006    6.2354       6.1214
log2 = 16, bandwidth = 1/8, validation: fails sev skew, agg skew.
```

(continues on next page)

(continued from previous page)

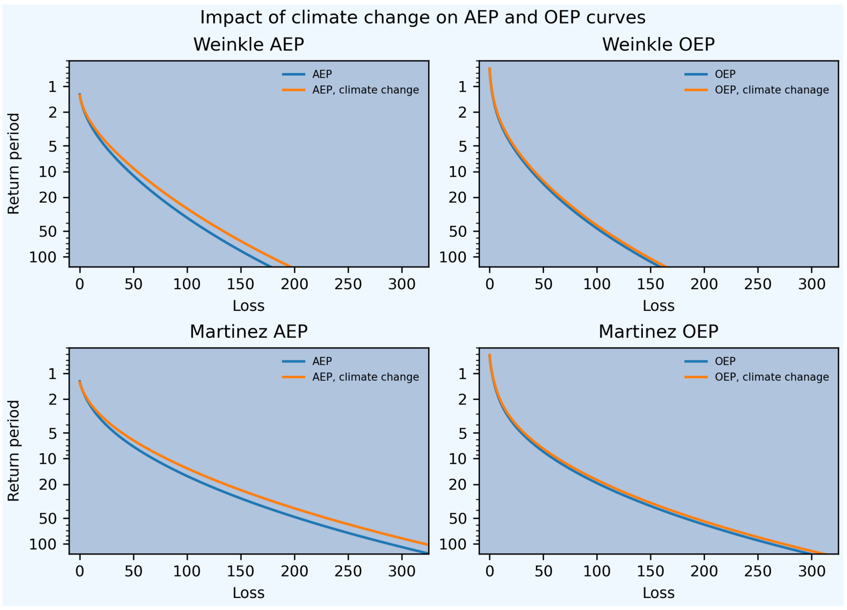
```
In [37]: mv(wcc)
mean     = 19.1129
variance = 1470.276
std dev  = 38.3442

In [38]: qd(mcc)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 1.7954                      0.86578             1.1454
Sev   16.95   16.949 -4.9432e-05  2.9533    2.9491   14.535      13.891
Agg  30.432   30.431 -4.9442e-05   2.368    2.3651   9.6255      9.2342
log2 = 16, bandwidth = 1/8, validation: fails sev cv, agg cv.

In [39]: mv(mcc)
mean     = 30.4321
variance = 5193.212
std dev  = 72.0639
```

The new models produce the following AALs, compare Jewson Figure 3.

```
In [40]: base = pd.concat((w.report_df.loc['agg_m'].T,
   ....:             m.report_df.loc['agg_m'].T), axis=1,
   ....:            keys=['Weinkle', 'Martinez']); \
   ....: cc = pd.concat((wcc.report_df.loc['agg_m'].T,
   ....:             mcc.report_df.loc['agg_m'].T), axis=1,
   ....:            keys=['Weinkle', 'Martinez']); \
   ....: df = pd.concat((base, cc), axis=1,
   ....:                keys=[' Base', 'Adjusted']); \
   ....: df[('Change', 'Martinez')] = (df[('Adjusted', 'Martinez')] -  df[(' Base',
↪ 'Martinez')]); \
   ....: df[('Change', 'Weinkle')] = (df[('Adjusted', 'Weinkle')] -  df[(' Base',
↪'Weinkle')]); \
   ....: df[('Pct Change', 'Martinez')] = (df[('Adjusted', 'Martinez')] -
   ....:                     df[(' Base', 'Martinez')]) / jewson.iloc[0]['EX_M']; \
   ....: df[('Pct Change', 'Weinkle')] = (df[('Adjusted', 'Weinkle')] -
   ....:                     df[(' Base', 'Weinkle')]) / jewson.iloc[0]['EX_W']; \
   ....: df = df.iloc[[0,1,2,3,4,6]]; \
   ....: df.index = [1,2,3,4,5, 'Total']; \
   ....: df.index.name = 'Category'; \
   ....: df = df.swaplevel(axis=1); \
   ....: df = df.sort_index(axis=1, ascending=[False, True]); \
   ....: qd(df.stack(0).swaplevel(0).sort_index(ascending=[False,True]))
   ....:


                    Base Adjusted   Change Pct Change
         Category
Weinkle  1        1.6188   1.6366 0.017807  0.0010663
         2         1.784   1.9535  0.16948   0.010149
         3          4.68   5.3071  0.62712   0.037552
         4         7.446   8.7788   1.3328    0.07981
         5        1.1625   1.4368  0.27435   0.016428
         Total    16.691   19.113   2.4216    0.14501
Martinez 1        2.1016   2.1247 0.023118 0.00087236
         2         2.556   2.7988  0.24282   0.009163
         3         6.444   7.3075   0.8635   0.032585
         4        13.991   16.495   2.5044   0.094505
         5          1.38   1.7057  0.32568    0.01229
         Total    26.473   30.432   3.9595    0.14942
```

Here are plots of the base and adjusted AEP and OEP curves. Compare Jewson Figure 5 (a) and (b) for aggregate and Figure 6 (a) and (b) for occurrence.

```
In [41]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.5), constrained_
↪layout=True)

In [42]: axs = axs.flat[::-1]

In [43]: for axo, axa, (mw, mwcc), title in zip(axs.flat[0::2], axs.flat[1::2],␣
↪[(m, mcc), (w, wcc)], ['Martinez', 'Weinkle']):
   ....:     bit = 1 / ((1 - mw.density_df.F_sev.loc[:2000]) * mw.n)
   ....:     bit.plot(logy=True, ax=axo, label='OEP');
   ....:     bit = 1 / ((1 - mwcc.density_df.F_sev.loc[:2000]) * mw.n)
   ....:     bit.plot(logy=True, ax=axo, label='OEP, climate chanage');
   ....:     bit = 1 / (1 - mw.density_df.F.loc[:2000])
   ....:     bit.plot(logy=True, ax=axa, label='AEP');
   ....:     bit = 1 / (1 - mwcc.density_df.F.loc[:2000])
   ....:     bit.plot(logy=True, ax=axa, label='AEP, climate change');
   ....:     axo.set(title=f'{title} OEP');
   ....:     axa.set(title=f'{title} AEP');
   ....:

In [44]: for ax in axs.flat:
   ....:     ax.set(xlim=[-10, 325], ylim=[130, .5], xlabel='Loss');
   ....:     if ax in [axs.flat[1], axs.flat[3]]:
   ....:         ax.set(ylabel='Return period');
   ....:     ax.set_yscale('log', base=10);
   ....:     ticks = [1, 2, 5, 10, 20, 50, 100]
   ....:     ax.set_yticks(ticks);
   ....:     ax.set_yticklabels([f'{x}' for x in ticks]);
   ....:     ax.legend()
   ....:

In [45]: fig.suptitle('Impact of climate change on AEP and OEP curves');
```



2.8.4 ILW Pricing

Industry Loss Warranties (ILW) are securities that pay an agreed amount if losses from a named peril exceed a threshold during the contract term. They are usually written on an occurrence basis and are triggered by losses from a single event. For example, a US hurricane $20 billion ILW pays 1 if there is a US hurricane causing $20 billion or more losses during the contract period. They are used by insurers to provide cat capacity. Because they are not written on an indemnity basis there is no underwriting, which simplifies their pricing.

Brokers publish price sheets for ILWs to give a view of market pricing. Price is expressed as a percentage of the face value. A recent sheet quoted prices for US hurricane as follows.

Attachment	Price (Pct)
$15B$	47.0
$20B$	38.0
$25B$	33.0
$30B$	27.5
$40B$	17.5
$50B$	13.0
$60B$	10.75

The next dataframe adds expected losses and compares them to the ILW pricing. The expected loss is given by the occurrence survival function — it is simply the probability of attaching the layer. The `EL` columns show Jewson's expected losses across the four views discussed above. The impact on EL is only caused by greater event frequency. Its effect increases with attachment.

```
In [46]: views = ['Weinkle', 'Weinkle Adj', 'Martinez', 'Martinez Adj']

In [47]: ilw = pd.concat((x.density_df.loc[[15, 20, 25, 30, 40, 50, 60],
   ....:                          ['S_sev']].rename(columns={'S_sev': 'EL'})
   ....:                          for x in [w, wcc, m, mcc]),
   ....:                     axis=1, keys=views,
   ....:                     names=['View', 'Stat']); \
   ....: ilw['Price'] = [.47, .38, .33, .275, .175, .13, .1075]; \
   ....: ilw.index.name = 'Trigger'; \
   ....: ilw = ilw.iloc[:, [-1,0,1,2,3]]; \
   ....: qd(ilw, float_format=lambda x: f'{x:.4f}')
   ....:

View     Price Weinkle Weinkle Adj Martinez Martinez Adj
Stat                EL          EL       EL           EL
Trigger
15.0    0.4700  0.1622      0.1733   0.2075       0.2207
20.0    0.3800  0.1261      0.1353   0.1666       0.1782
25.0    0.3300  0.1013      0.1091   0.1390       0.1492
30.0    0.2750  0.0832      0.0898   0.1188       0.1279
40.0    0.1750  0.0586      0.0634   0.0910       0.0983
50.0    0.1300  0.0431      0.0467   0.0725       0.0785
60.0    0.1075  0.0326      0.0354   0.0593       0.0643
```

Cat pricing is often expressed in terms of the implied multiple: the ratio of premium to EL (reciprocal of the loss ratio).

Cat pricing multiples are usually in the range of 2 to 5 times the standard commercial models' estimates of expected loss. The base model pricing multiples, shown below, fall into this range. The climate adjusted multiples fall outside, which is not unexpected, since the pricing is for the coming period and not a future climate-impacted period.

```
In [48]: ilw[[(v, 'Multiple') for v in views]] = ilw[['Price']].values / ilw[[(v,
↪'EL') for v in views]]; \
   ....: qd(ilw.iloc[:, [0,5,6,7,8]])
   ....:

View     Price  Weinkle Weinkle Adj Martinez Martinez Adj
```

(continues on next page)

(continued from previous page)

```
Stat             Multiple    Multiple Multiple     Multiple
Trigger
15.0       0.47   2.8979      2.7117   2.2652       2.1299
20.0       0.38   3.0142      2.8079   2.2803       2.1327
25.0       0.33   3.2578      3.0257   2.3739       2.2117
30.0      0.275   3.3058      3.0637   2.3144       2.1502
40.0      0.175   2.9845      2.7581   1.9239       1.7806
50.0       0.13   3.0188      2.7849   1.7938       1.6562
60.0     0.1075   3.3001      3.0409   1.8132       1.6714
```

The next table shows implied distortion parameters calibrated to market pricing for the dual

$$g(s) = 1 - (1 - s)^p,\ p > 1$$

and proportional hazard (PH)

$$g(s) = s^p,\ p < 1$$

parametric families (see Mildenhall and Major [2022]). In both cases, a higher parameter corresponds to a higher risk load. The dual is body-risk centric and the PH is tail-risk centric. The indicated parameters are quite high, consistent with the expense of bearing cat risk. (The parameters are incomparable between distortions.)

```
In [49]: params = pd.concat((np.log(1 - ilw[['Price']]).values / np.log(1 - ilw.xs(
↪'EL', axis=1, level=1)),
   ....:                  np.log(ilw[['Price']].values) / np.log(ilw.xs('EL',␣
↪axis=1, level=1))),
   ....:                  axis=1, keys=['Dual', 'PH'])
   ....:

In [50]: qd(params.xs('Dual', axis=1, level=0))

View     Weinkle  Weinkle Adj  Martinez  Martinez Adj
Trigger
15.0      3.5877       3.3355    2.7301        2.5465
20.0      3.5474       3.2875    2.6224         2.436
25.0      3.7497       3.4678    2.6757        2.4785
30.0      3.7027       3.4194    2.5422        2.3499
40.0      3.1837       2.9347    2.0172        1.8596
50.0      3.1637       2.9131    1.8511        1.7036
60.0      3.4341       3.1598    1.8608        1.7107

In [51]: qd(params.xs('PH', axis=1, level=0))

View     Weinkle  Weinkle Adj  Martinez  Martinez Adj
Trigger
15.0     0.41507       0.4308   0.48009       0.49965
20.0     0.46722      0.48378   0.53997       0.56093
25.0      0.4842      0.50034   0.56186       0.58276
30.0     0.51916      0.53554   0.60606       0.62775
40.0      0.6145      0.63208   0.72704        0.7513
50.0      0.6487      0.66578   0.77736       0.80174
60.0     0.65132      0.66726   0.78937        0.8128
```

2.8.5 Secondary Uncertainty

Secondary uncertainty is the practice of expanding cat model simulated output by assuming that the results from each event form a distribution. It is usual to assume the distribution is a beta. The model output provides the beta's mean and standard deviation. Given this output, modelers often need to compute statistics, such as a layer expected loss, reflecting the secondary uncertainty. This calculation can be performed in `aggergate` as follows.

Assumptions: Assume one location with a TIV of 2500 and simple cat model output with only three equally-likely events with mean losses 100, 200, and 1100 and secondary uncertainty standard deviation 100, 150, and 600. The overall event frequency is 1.6 with a Poisson distribution.

Question: What is the expected loss to a 1000 xs 1000 per risk cover with and without secondary uncertainty?

Solution: Start by building the answer without secondary uncertainty. It is convenient to put the assumptions in a dataframe.

```
In [52]: df = pd.DataFrame({'GroundUpLoss': [100, 200, 1100],
   ....:                    'GroundUpSD': [100, 150, 600]})
   ....:
```

The model with no secondary uncertainty is a simple mixed severity.

```
In [53]: base = build('agg Cat:Base '
   ....:              '1.6 claims '
   ....:              f'dsev {df.GroundUpLoss.values} '
   ....:              'occurrence ceded to 1000 xs 1000 '
   ....:              'poisson'
   ....:              , bs=1)
   ....:

In [54]: qd(base)

       E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    1.6                   0.79057            0.79057
Sev   466.67   33.333 -0.92857 0.96362    1.4142  0.68097     0.70711
Agg   746.67   53.333 -0.92857  1.0979    1.3693   1.2973      1.3693
log2 = 15, bandwidth = 1, validation: n/a, reinsurance.
```

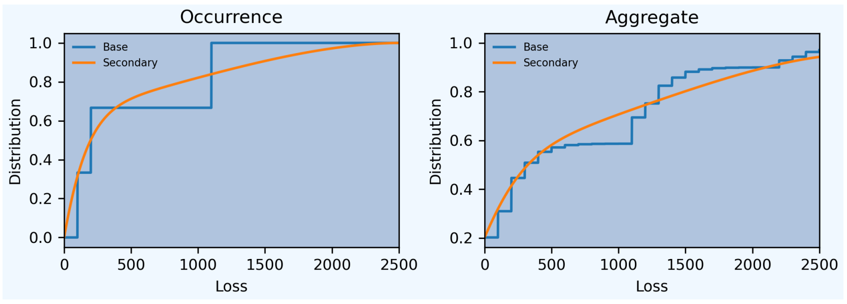
To incorporate the secondary uncertainty, we first compute the beta parameters using the method of moments. Then build the `Aggregate` model incorporating secondary uncertainty in each loss.

```
In [55]: tiv = 2500;                                \
   ....: m = df['GroundUpLoss'] / tiv;              \
   ....: v = (df['GroundUpSD'] / tiv) ** 2;         \
   ....: sev_a = m * (m * (1 - m) / v - 1);         \
   ....: sev_b = (1 - m) * (m * (1 - m) / v - 1);   \
   ....: sec = build(f'agg Cat:Secondary '
   ....:             '1.6 claims '
   ....:             f'sev {tiv} * beta {sev_a.values} {sev_b.values} wts=3 '
   ....:             'occurrence ceded to 1000 xs 1000 '
   ....:             'poisson')
   ....:

In [56]: qd(sec)

       E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    1.6                   0.79057            0.79057
Sev   466.67   96.384 -0.79346  1.2367    2.5758   1.5399      2.6259
Agg   746.67   154.21 -0.79346  1.2573    2.1844   1.6706      2.4651
log2 = 16, bandwidth = 1/2, validation: n/a, reinsurance.
```

Including secondary uncertainty nearly triples the expected loss to the layer, from 53 to 154. Had the third loss been only 1000, there would be no loss at all to the layer without secondary uncertainty.

The next plot compares the gross severity and aggregate distributions.

```
In [57]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True); \
   ....: ax0, ax1 = axs.flat
   ....:

In [58]: base.reinsurance_df['p_sev_gross'].cumsum().plot(xlim=[0, 2500], ax=ax0,␣
↪label='Base'); \
   ....: sec.reinsurance_df['p_sev_gross'].cumsum().plot(xlim=[0, 2500], ax=ax0,␣
↪label='Secondary'); \
   ....: base.reinsurance_df['p_agg_gross_occ'].cumsum().plot(xlim=[0, 2500],␣
↪ax=ax1, label='Base'); \
   ....: sec.reinsurance_df['p_agg_gross_occ'].cumsum().plot(xlim=[0, 2500],␣
↪ax=ax1, label='Secondary'); \
   ....: ax0.set(title='Occurrence', xlabel='Loss', ylabel='Distribution'); \
   ....: ax1.set(title='Aggregate', xlabel='Loss', ylabel='Distribution'); \
   ....: ax0.legend();
   ....:

In [59]: ax1.legend();
```



2.8.6 Summary of Objects Created by DecL

The following objects are created by `build()` in this guide.

```
In [60]: from aggregate import pprint_ex

In [61]: for n, r in build.qlist('^Cat:').iterrows():
   ....:     pprint_ex(r.program, split=20)
   ....:
```

2.9 Capital Modeling and Risk Management

Objectives: Application of the `Portfolio` class to capital modeling, including VaR, TVaR, and risk visualization and quantification. Covers material on CAS Part 9.

Audience: ERM, capital modeling, risk management actuaries.

Prerequisites: DecL, aggregate distributions, risk measures.

See also: *Catastrophe Modeling*, *Strategy and Portfolio Management*, *Case Studies*.

Contents:

1. Helpful References
2. *Conditional Expectation as a Risk Management and Visualization Device*

2.9.1 Helpful References

- Mildenhall and Major [2022], especially chapter 14.

2.9.2 Conditional Expectation as a Risk Management and Visualization Device

Todo: Documentation to follow. In the meantime, see examples in *Case Studies*.

2.10 Strategy and Portfolio Management

Objectives: Use spectral risk measures to allocate total margin by unit.Application of the `Portfolio` and and `Distortion` classes to strategy and portfolio management, including margin (capital) allocation, determining benchmark pricing within a portfolio using alternative pricing methodologies, and the evaluation of reinsurance.

Audience: Planning and strategy, ERM, capital modeling, risk management actuaries.

Prerequisites: DecL, aggregate distributions, risk measures.

See also: *Catastrophe Modeling*, *Capital Modeling and Risk Management*, *Case Studies*.

Contents:

2.10.1 Helpful References

- Mildenhall and Major [2022] chapters 14 and 15 and references therein.

2.10.2 Margin Allocation Using Spectral Risk Measures

Todo: Documentation to follow. In the meantime, see examples in *Case Studies*.

2.11 Case Studies

Objectives: Using `aggregate` to reproduce the case study exhibits from the book Pricing Insurance Risk and build similar exhibits for your own cases.

Audience: Capital modeling and corporate strategy actuaries; anyone reading PIR.

Prerequisites: Intermediate to advanced users with a sold understanding of `aggregate`. Familiar with PIR.

See also: *Capital Modeling and Risk Management*, *Strategy and Portfolio Management*.

Contents:

5. *Standard Case Study Exhibits*

Confession: Some of the `case_studies` code is sub-optimal.

2.11.1 PIR Case Studies

The book Pricing Insurance Risk (PIR) presents four Case Studies that show how different methods price business. This section shows how to reproduce all the book's exhibits for each case and how to create new cases.

Each case describes business written by Ins Co., a one-period de novo insurer that comes into existence at time zero, raises capital and writes business, and pays all losses at time one. A case models two units (line, region, operating unit, or other division) with one more risky than the other. Usually, the riskier one is reinsured. Case exhibits compare unit statistics and pricing on a gross and net basis, showing results from over a dozen different methods.

Simple Discrete Example

In the Simple Discrete Example Case Study, X_1 takes values 0, 8, or 10, and X_2 values 0, 1, or 90. The units are independent and sum to the portfolio loss X. The outcome probabilities are 1/2, 1/4, and 1/4 respectively for each marginal. There are 9 possible outcomes. This type of output is typical of that produced by a catastrophe, capital, or pricing simulation model—albeit much simpler.

Tame Case Study

In the Tame Case Study, Ins Co. writes two predictable units with no catastrophe exposure. It is an idealized, best-case risk-pool. It proxies a portfolio of personal and commercial auto liability. It uses a straightforward stochastic model with gamma distributions.

Aggregate reinsurance applies to the more volatile unit, with an attachment probability 0.2 (56) and detachment probability 0.01 (69).

Catastrophe and Non-Catastrophe (CNC) Case Study

In the Cat/Non-Cat Case Study, Ins Co. has catastrophe and non-catastrophe exposures. The non-catastrophe unit proxies a small commercial lines portfolio. Balancing the relative benefits of units considered to be more stable against more volatile ones is a very common strategic problem for insurers and reinsurers. It arises in many different guises:

- Should a US Midwestern company expand to the East coast (and pick up hurricane exposure)?
- Should an auto insurer start writing homeowners?
- What is the appropriate mix between property catastrophe and non-catastrophe exposed business for a reinsurer?

The two units are independent and have gamma and lognormal distributions.

Aggregate reinsurance applies to the Cat unit, with an attachment probability 0.1 (41) and detachment probability 0.005 (121).

Hurricane/Severe Convective Storm (HuSCS) Case Study

In the Hu/SCS Case Study, Ins Co. has catastrophe exposures from severe convective storms (SCS) and, independently, hurricanes (Hu). In practice, hurricane exposure is modeled using a catastrophe model. The Case proxies that by using a very severe severity in place of the gross catastrophe model event-level output. Both units are modeled using an aggregate distribution with a Poisson frequency and lognormal severity.

Aggregate reinsurance applies to the HU unit with an occurrence attachment probability 0.05(40) and detachment probability 0.005 (413). (PIR incorrectly states the reinsurance is occurrence.)

2.11.2 Creating a Case Study

Case Study exhibits are managed by the class `CaseStudy` in `aggregate.extensions.case_studies`, see *Extensions*. Here are the steps needed to create a case study. The computations take a few minutes. The output is a set of HTML files that can be viewed in a browser. The code blocks below are provided in executable scripts described below.

1. Import `case_studies`:

```
from aggregate import build, qd
from aggregate.extensions import case_studies as cs
```

2. Create a new `CaseStudy` object. It is a generic container, the options are set in the next step:

```
my_case = cs.CaseStudy()
```

3. Set the case study options. Here are the options for the PIR Tame case. The arguments are described later:

```
my_case.factory(case_id='my_tame',
                case_name='My version of PIR Tame Case',
                case_description='Tame Case to demonstrate capabilities.',
                a_distribution='agg A 1 claim sev gamma 50 cv 0.10 fixed',
                b_distribution_gross='agg B 1 claim sev gamma 50 cv 0.15 fixed
↪',
                b_distribution_net='agg B 1 claim sev gamma 50 cv 0.15 fixed '
                                   'aggregate net of 12.90625 xs 56.171875',
                reg_p=0.999,
                roe=0.10,
                d2tc=0.3,
                f_discrete=False,
                s_values=[.005, 0.01, 0.03],
                gs_values=[0.029126,   0.047619,   0.074074],
                bs=1/64,
                log2=16,
                padding=1)
```

4. Execute:

```
my_case.full_monty()
```

to create all figures and tables. The code can take several minutes to execute.

5. To browse the exhibits execute:

```
my_case.browse_exhibits()
```

which opens two new browser tabs, one for the standard book exhibits and one for a set of extended exhibits.

Details.

Exhibit output files are stored in `build.case_dir`, which by default is the subdirectory aggregate/cases of your home directory (~ on Linux, `\users\<user name>` on Windows, and who knows on an Apple). The book exhibits are marshaled in `f'{my_case.case_id}_book.html'` and a set of extended exhibits are in `f'{my_case.case_id}_extended.html'`. The detailed files are in a subdirectory called `my_case.case_id`.

2.11.3 Case Study Factory Arguments

`cs.CaseStudy().factory()` takes the following arguments.

- `case_id` is a single word label that uniquely identifies the Case. It determines the output directory for the Case exhibits and so must be acceptable to your operating system as a directory name.
- `case_name` such as "Cat/Non-Cat".
- `case_description` is a brief description of the Case.
- `a_distribution`, `b_distribution_gross` and `b_distribution_net` are DecL programs defining the aggregate distributions for each unit, including reinsurance on unit B. Unit names should be upper case and ideally in alphabetical order, the more volatile unit second. For example `A` and `B`.
- `reg_p` gives the regulatory capital standard, entered as a probability of non-exceedance level. Solvency II operates at 0.995 (one in 200 years). In the US, rating agencies consider companies at 0.99 (100 years), 0.996 (250 years), 0.999 (1000 years), or higher. Corporate bond default rates impose even tighter capital standards.
- `roe` sets the target cost of capital. All pricing methods are calibrated to produce a return on capital of `roe` at the selected capital standard level. This makes them comparable.
- `d2tc` is the maximum allowable debt to total capital level (used for enhanced exhibits only). It is used to tranche capital into debt and equity.
- `f_discrete` indicates whether the distributions are discrete (Simple Discrete Example) or mixed (all others). Usually `f_discrete=False`.
- `s_values` and `gs_values` define cat bond pricing and are used to create a blended distortion, see below.
- `bs`, `log2`, and `padding` are the usual update parameters.

2.11.4 PIR Case Specifications

This section provides the arguments needed to recreate each PIR Case Study. Some of the code is used to determine the details of reinsurance. See PIR Chapter REF for more details and explanation.

The current version of `aggregate` uses an improved blend distortion over that shown in PIR. It is calibrated to cat bond pricing for high return periods using the following values:

- `s_values: [.005, 0.01, 0.03]`
- `gs_values: [0.029126, 0.047619, 0.074074]`

meaning bonds with a 0.5% EL have a discount spread of 2.9%, 1% EL a discount spread of 4.76% and so forth. They define the left-hand (small s) end of a distortion function.

All Cases assume that debt to total capital limited at 30%, `d2tc: 0.3`. This factor is only used in the extended exhibits.

Simple Discrete Example Specification

There are two flavors, one with distinct outcomes and one with two ways of obtaining the outcome 10. The latter is used in PIR to illustrate the linear natural allocation. Here are the specifications.

```
# make the discrete case study

from aggregate.extensions import case_studies as cs

if __name__ == '__main__':

    discrete = cs.CaseStudy()
    discrete.factory(case_id='discrete',
                     case_name='Discrete',
                     case_description='PIR Discrete Case Study (no equal points).',
                     a_distribution='agg X1 1 claim dsev [0 8 10] [1/2 1/4 1/4]␣
↪fixed',
                     b_distribution_gross='agg X2 1 claim dsev [0 1 90] [1/2 1/4 1/
↪4] fixed',
                     b_distribution_net=f'agg X2 1 claim dsev [0 1 90] [1/2 1/4 1/
↪4] fixed aggregate net of 70 xs 20',
                     reg_p=1,
                     roe=0.10,
                     d2tc=0.3,
                     f_discrete=True,
                     s_values=[.005, 0.01, 0.03],
                     gs_values=[0.029126,   0.047619,   0.074074],
                     bs=1,
                     log2=8,
                     padding=1)
    discrete.full_monty()
    discrete.to_json()
    discrete.browse_exhibits()

    discrete_eq = cs.CaseStudy()
    discrete_eq.factory(case_id='discrete_equal',
                        case_name='Discrete (equal points)',
                        case_description='PIR Discrete Case Study with equal␣
↪points.',
                        a_distribution='agg X1 1 claim dsev [0 9 10] [1/2 1/4 1/4]␣
↪fixed',
                        b_distribution_gross='agg X2 1 claim dsev [0 1 90] [1/2 1/
↪4 1/4] fixed',
                        b_distribution_net=f'agg X2 1 claim dsev [0 1 90] [1/2 1/4␣
↪1/4] fixed aggregate net of 70 xs 20',
                        reg_p=1,
                        roe=0.10,
                        d2tc=0.3,
                        f_discrete=True,
                        s_values=[.005, 0.01, 0.03],
                        gs_values=[0.029126,   0.047619,   0.074074],
                        bs=1,
                        log2=8,
                        padding=1)
    discrete_eq.full_monty()
    discrete_eq.to_json()
    discrete_eq.browse_exhibits()
```

Tame Specification

The first few lines calibrate the reinsurance to probability levels.

```
# make the tame case study

from aggregate.extensions import case_studies as cs
from aggregate import build

if __name__ == '__main__':

    # calibrate reinsurance
    recalc = build('agg B 1 claim sev gamma 50 cv 0.15 fixed',
                   log2=16, bs=1/64)
    a, d = recalc.q(0.8), recalc.q(0.99)
    y = d - a

    # create exhibits
    tame = cs.CaseStudy()
    tame.factory(case_id='tame',
                 case_name='Tame',
                 case_description='Tame Case in the new syntax.',
                 a_distribution='agg A 1 claim sev gamma  50 cv 0.10 fixed',
                 b_distribution_gross='agg B 1 claim sev gamma  50 cv 0.15 fixed',
                 b_distribution_net=f'agg B 1 claim sev gamma  50 cv 0.15 fixed␣
↪aggregate net of {y} xs {a}',
                 reg_p=0.9999,
                 roe=0.10,
                 d2tc=0.3,
                 f_discrete=False,
                 s_values=[.005, 0.01, 0.03],
                 gs_values=[0.029126,   0.047619,   0.074074],
                 bs=1/64,
                 log2=16,
                 padding=1)
    tame.full_monty()
    tame.to_json()
    tame.browse_exhibits()
```

Catastrophe and Non-Catastrophe Specification

The first few lines calibrate the reinsurance to probability levels.

```
# make the cat non-cat case study

from aggregate.extensions import case_studies as cs
from aggregate import build
import warnings

warnings.filterwarnings('ignore')

if __name__ == '__main__':
    # be more informative
    build.logger_level(20)
    # calibrate reinsurance
    recalc = build('agg Cat 1 claim sev lognorm 20 cv 1.00 fixed'
                   , log2=16, bs=1/64)
    a, d = recalc.q(0.9), recalc.q(0.995)
    y = d - a

    # create exhibits
```

(continues on next page)

(continued from previous page)

```
    cnc = cs.CaseStudy()
    cnc.factory(case_id='cnc',
                case_name='Cat/Non-Cat',
                case_description='Cat/Non-Cat in the new syntax.',
                a_distribution='agg NonCat 1 claim sev gamma 80 cv 0.15 fixed',
                b_distribution_gross='agg Cat 1 claim sev lognorm 20 cv 1.00 fixed
↪',
                b_distribution_net=f'agg Cat 1 claim sev lognorm 20 cv 1.00 fixed␣
↪aggregate net of {y} xs {a}',
                reg_p=0.999,
                roe=0.10,
                d2tc=0.3,
                f_discrete=False,
                s_values=[.005, 0.01, 0.03],
                gs_values=[0.029126,   0.047619,   0.074074],
                bs=1/64,
                log2=16,
                padding=1)
    cnc.full_monty()
    cnc.to_json()
    cnc.browse_exhibits()
```

Hurricane/Severe Convective Storm Specification

The first few lines set up the parameters from PIR. Then the reinsurance is calibrated to probability levels. The second version has occurrence reinsurance on the Cat line with the same limit and attachment.

```
# make the hu-scs case study

from aggregate.extensions import case_studies as cs
from aggregate import build
import numpy as np

if __name__ == '__main__':

      # parameters from PIR
      freq1, sigma1 = 70, 1.9
      freq2, sigma2 = 2, 2.5
      mu1 = -sigma1**2 / 2
      mu2 = -sigma2**2 / 2
      sev1, sev2 = 1, 15
      print(mu1, sigma1, mu2, sigma2,
            freq1, np.exp(mu1), sigma1,
            freq2, np.exp(mu2), sigma2)

      recalc = build(f'sev Husev {15 * np.exp(mu2)} * lognorm {sigma2}')
      a, d = recalc.isf(0.05), recalc.isf(0.005)
      y = d - a
      print(y, a, d)

      # create exhibits
      hs = cs.CaseStudy()
      hs.factory(case_id='hs',
                 case_name='Hu/SCS Case',
                 case_description='Hu/SCS Case in the new syntax.',
                 a_distribution=f'agg SCS {freq1} claims sev {sev1 * np.exp(mu1)}␣
↪* lognorm {sigma1} poisson',
                 b_distribution_gross=f'agg Hu {freq2} claims sev {sev2 * np.
↪exp(mu2)} * lognorm {sigma2} poisson',
```

(continues on next page)

(continued from previous page)

```
                 b_distribution_net=f'agg Hu {freq2} claims sev {sev2 * np.
↪exp(mu2)} * lognorm {sigma2} poisson ' \
                                   f'aggregate net of {y} xs {a}',
                 reg_p=0.999,
                 roe=0.10,
                 d2tc=0.3,
                 f_discrete=False,
                 s_values=[.005, 0.01, 0.03],
                 gs_values=[0.029126,   0.047619,   0.074074],
                 bs=1/4,
                 log2=19,
                 padding=1)
      hs.full_monty()
      hs.to_json()
      hs.browse_exhibits()


      hs2 = cs.CaseStudy()
      hs2.factory(case_id='hs_per_occ',
                  case_name='Hu/SCS',
                  case_description='Hu/SCS Case in the new syntax with per␣
↪occurrence reinsurance .',
                  a_distribution=f'agg SCS {freq1} claims sev {sev1 * np.exp(mu1)}␣
↪ * lognorm {sigma1} poisson',
                  b_distribution_gross=f'agg Hu {freq2} claims sev {sev2 * np.
↪exp(mu2)} * lognorm {sigma2} poisson',
                  b_distribution_net=f'agg Hu {freq2} claims sev {sev2 * np.
↪exp(mu2)} * lognorm {sigma2} '
                                    f'occurrence net of {y} xs {a} poisson',
                  reg_p=0.999,
                  roe=0.10,
                  d2tc=0.3,
                  f_discrete=False,
                  s_values=[.005, 0.01, 0.03],
                  gs_values=[0.029126,   0.047619,   0.074074],
                  bs=1/4,
                  log2=19,
                  padding=1)
      hs2.full_monty()
      hs2.to_json()
```

These snippets are provided in Python scripts in `aggregate.extensions` called `discrete.py`, `tame.py`, `cnc.py` and `hs.py`. They can be run from the command line:

```
python -m aggregate.extensions.tame
```

Precomputed versions are available at https://www.pricinginsurancerisk.com/results.

2.11.5 Defining a Custom Case Study

It should be obvious how to create a custom case study. The key is the DecL for the two units. Here are ideas for some custom Case Studies illustrating different behaviors.

2.11.6 Standard Case Study Exhibits

The next table provides a list of all the PIR exhibits and figures showing the Chapter in which they occur and the figure numbers. Not all exhibits are down for the Simple Discrete Example.

Rf	Kind	Ch.	Number(s)	Description
A	Table	2	2.3, 2.5, 2.6, 2.7	Estimated mean, CV, skewness and kurtosis by line and in total, gross and net.
B	Figure	2	2.2, 2.4, 2.6	Gross and net densities on a linear and log scale.
C	Figure	2	2.3, 2.5, 2.7	Bivariate densities: gross and net with gross sample.
D	Figure	4	4.9, 4.10, 4.11, 4.12	TVaR, and VaR for unlimited and limited variables, gross and net.
E	Table	4	4.6, 4.7, 4.8	Estimated VaR, TVaR, and EPD by line and in total, gross, and net.
F	Table	7	7.2	Pricing summary.
G	Table	7	7.3	Details of reinsurance.
H	Table	9	9.2, 9.5, 9.8	Classical pricing by method.
I	Table	9	9.3, 9.6, 9.9	Sum of parts (SoP) stand-alone vs. diversified classical pricing by method.
J	Table	9	9.4, 9.7, 9.10	Implied loss ratios from classical pricing by method.
K	Table	9	9.11	Comparison of stand-alone and sum of parts premium.
L	Table	9	9.12, 9.13, 9.14	Constant CoC pricing by unit for Case Study.
M	Figure	11	11.2, 11.3, 11.4,11.5	Distortion envelope for Case Study, gross.
N	Table	11	11.5	Parameters for the six SRMs and associated distortions.
O	Figure	11	11.6, 11.7, 11.8	Variation in insurance statistics for six distortions as *s* varies.
P	Figure	11	11.9, 11.10, 11.11	Variation in insurance statistics as the asset limit is varied.
Q	Table	11	11.7, 11.8, 11.9	Pricing by unit and distortion for Case Study.
R	Table	13	13.1 missing	Comparison of gross expected losses by Case, catastrophe-prone lines.
S	Table	13	13.2, 13.3, 13.4	Constant 0.10 ROE pricing for Case Study, classical PCP methods.
T	Figure	15	15.2 - 15.7 (G/N)	Twelve plot.
U	Figure	15	15.8, 15.9, 15.10	Capital density by layer.
V	Table	15	15.35, 15.36, 15.37	Constant 0.10 ROE pricing for Cat/Non-Cat Case Study, distortion, SRM methods.
W	Figure	15	15.11	Loss and loss spectrums.
X	Figure	15	15.12, 15.13, 15.14	Percentile layer of capital allocations by asset level.
Y	Table	15	15.38, 15.39, 15.40	Percentile layer of capital allocations compared to distortion allocations.

2.12 Working With Samples

Objectives: How to sample from `aggregate` and how to a build a `Portfolio` from a sample. Inducing correlation in a sample using the Iman-Conover algorithm and determining the worst-VaR rearrangement using the rearrangement algorithm.

Audience: Planning and strategy, ERM, capital modeling, risk management actuaries.

Prerequisites: DecL, aggregate distributions, risk measures.

See also: ../5_technical_guides/5_x_samples, ../5_technical_guides/5_x_iman_conover, ../5_technical_guides/5_x_rearrangement_algorithm.

Contents:

2.12.1 Helpful References

- Mildenhall and Major [2022] chapter 14 and 15
- Puccetti and Ruschendorf [2012]
- Conover [1999]
- Mildenhall [2005]
- Vitale IC proof in dependency book

2.12.2 Samples and Densities

Use case: make realistic marginal distributions with `aggregate` that reflect the underlying frequency and severity (rather than defaulting to a lognormal determined by a CV assumption) and then use a sample in your simulation model.

2.12.3 Samples from `aggregate` Object

The method `sample()` draws a sample from an `Aggregate` or `Portfolio` class object. Both cases work by applying `pandas.DataFrame.sample` to the object's `density_df` dataframe.

Examples.

1. A sample from an `Aggregate`. Set up a simple lognormal distribution, modeled as an aggregate with trivial frequency.

```
In [1]: from aggregate import build, qd, set_seed

In [2]: a01 = build('agg Samp:01 '
   ...:             '1 claim '
   ...:             'sev lognorm 10 cv .4 '
   ...:             'fixed'
   ...:           , bs=1/512)
   ...:
```

(continues on next page)

(continued from previous page)

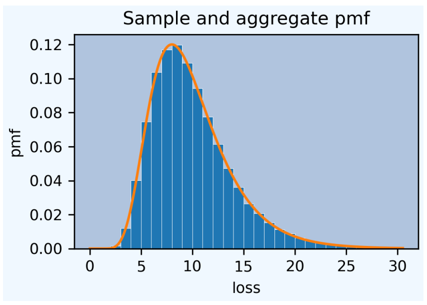
```
In [3]: qd(a01)

      E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
X
Freq     1                            0
Sev     10        10 -6.1087e-11     0.4        0.4    1.264        1.264
Agg     10        10 -6.1087e-11     0.4        0.4    1.264        1.264
log2 = 16, bandwidth = 1/512, validation: not unreasonable.
```

Apply `sample()` and display the results.

```
In [4]: set_seed(102)

In [5]: df = a01.sample(10**5)

In [6]: fc = lambda x: f'{x:8.2f}'

In [7]: qd(df.head(), float_format=fc)

      loss
0     6.33
1    10.10
2    13.29
3     7.56
4    11.18
```

The sample histogram and the computed pmf are close. The pmf is adjusted to the resolution of the histogram.

```
In [8]: fig, ax = plt.subplots(1, 1, figsize=(3.5, 2.45), constrained_
↪layout=True)

In [9]: xm = a01.q(0.999)

In [10]: df.hist(bins=np.arange(xm), ec='w', lw=.25, density=True,
   ....:         ax=ax, grid=False);
   ....:

In [11]: (a01.density_df.loc[:xm, 'p_total'] / a01.bs).plot(ax=ax);

In [12]: ax.set(title='Sample and aggregate pmf', ylabel='pmf');
```



2. A sample from a `Portfolio` produces a multivariate distribution. Setup a simple `Portfolio` with three lognormal marginals.

```
In [13]: from aggregate.utilities import qdp

In [14]: from pandas.plotting import scatter_matrix

In [15]: p02 = build('port Samp:02 '
```

(continues on next page)

(continued from previous page)

```
   ....:          'agg A 1 claim sev lognorm 10 cv .2 fixed '
   ....:          'agg B 1 claim sev lognorm 15 cv .5 fixed '
   ....:          'agg C 1 claim sev lognorm  5 cv .8 fixed '
   ....:         , bs=1/128)
   ....:

In [16]: qd(p02)

            E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est␣
↪Skew(X)
unit  X                                                             ␣
↪
A     Freq     1                            0                       ␣
↪
      Sev     10       10  5.7732e-15     0.2        0.2   0.608        0.
↪608
      Agg     10       10  5.7732e-15     0.2        0.2   0.608        0.
↪608
B     Freq     1                            0                       ␣
↪
      Sev     15       15 -2.2271e-13     0.5        0.5   1.625        1.
↪625
      Agg     15       15 -2.2271e-13     0.5        0.5   1.625        1.
↪625
C     Freq     1                            0                       ␣
↪
      Sev      5        5 -2.3183e-10     0.8        0.8   2.912        2.
↪912
      Agg      5        5 -2.3183e-10     0.8        0.8   2.912        2.
↪912
total Freq     3                            0                       ␣
↪
      Sev     10       10 -3.8748e-11 0.64872            1.8077         ␣
↪
      Agg     30       30 -6.3166e-11 0.29107    0.29107  1.3168       1.
↪3168
log2 = 16, bandwidth = 1/128, validation: not unreasonable.
```

Apply `sample()` to produce a sample with no correlation. Here are the first few values.

```
In [17]: df = p02.sample(10**4)

In [18]: qd(df.head(), float_format=fc)

          A        B        C    total
0      9.45    21.07     4.86    35.38
1     11.73    20.10     8.34    40.16
2     10.93    12.06     3.05    26.05
3      7.24     6.34     1.79    15.37
4      8.28    13.30    13.87    35.45
```

`qdp()` prints the pandas `describe` statistics dataframe for a dataframe, and adds the CV.

```
In [19]: qdp(df)
Out[19]:
                  A             B             C         total
count  10000.000000  10000.000000  10000.000000  10000.000000
mean       9.966709     15.052769      5.058216     30.077694
std        2.002445      7.583697      4.047945      8.889127
min        3.664062      2.117188      0.250000     11.726562
25%        8.539062      9.773438      2.484375     23.781250
```

(continues on next page)

(continued from previous page)

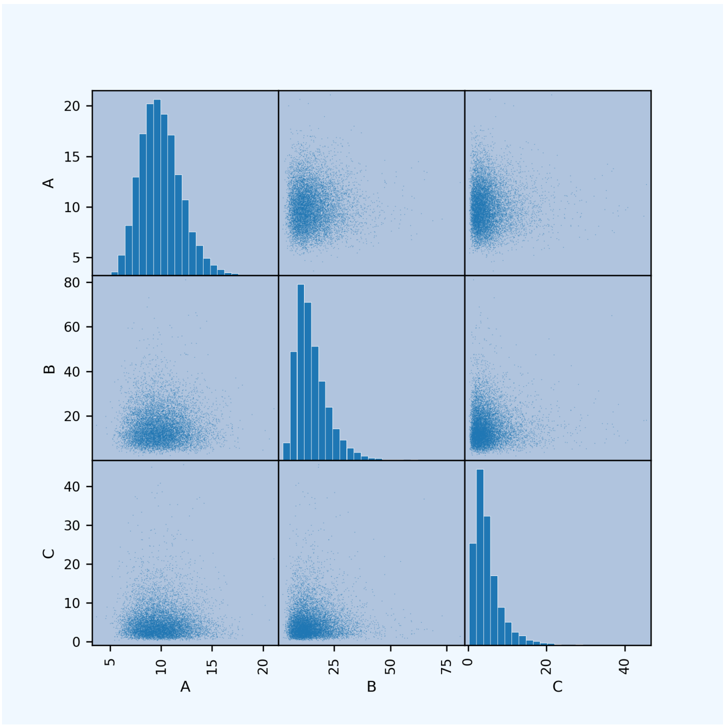
```
50%         9.789062     13.351562      4.000000     28.507812
75%        11.187500     18.609375      6.257812     34.640625
max        21.078125     81.070312     45.562500    115.281250
cv          0.200913      0.503807      0.800271      0.295539
```

The sample is independent, with correlations close to zero, as expected.

```
In [20]: abc = ['A', 'B', 'C']

In [21]: qd(df[abc].corr())

           A         B          C
A          1 0.0058516 -0.0021565
B  0.0058516         1   0.015729
C -0.0021565  0.015729          1
```

The scatterplot is consistent with independent marginals.

```
In [22]: scatter_matrix(df[abc], grid=False,
   ....:     figsize=(6, 6), diagonal='hist',
   ....:     hist_kwds={'density': True, 'bins': 25, 'lw': .25, 'ec': 'w'}
↪,
   ....:     s=1, marker='.');
   ....:
```



3. Pass a correlation matrix to `sample()` to draw a correlated sample. Correlation is induced using the Iman-Conover algorithm.

   The function `random_corr_matrix()` creates a random correlation matrix using vines. The second parameter controls the average correlation. This example includes high positive correlation.

```
In [23]: from aggregate import random_corr_matrix

In [24]: rcm = random_corr_matrix(3, .6, True)

In [25]: rcm
```

(continues on next page)

(continued from previous page)

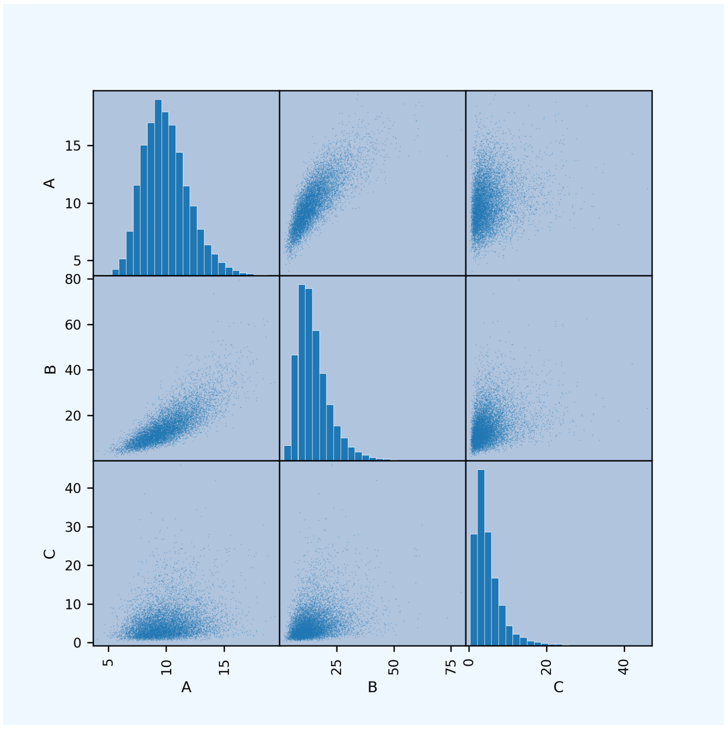
```
Out[25]:
matrix([[1.        , 0.80331094, 0.24377874],
        [0.80331094, 1.        , 0.35448898],
        [0.24377874, 0.35448898, 1.        ]])
```

Re-sample with target correlation `rcm`. The achieved correlation is reasonably close to the requested `rcm`.

```
In [26]: df2 = p02.sample(10**4, desired_correlation=rcm)

In [27]: qd(df2.iloc[:, :3].corr('pearson'))

        A       B       C
A       1 0.78513 0.22481
B 0.78513       1 0.31656
C 0.22481 0.31656       1
```

The scatterplot now shows correlated marginals. The histograms are unchanged.

```
In [28]: df2['total'] = df2.sum(1)

In [29]: scatter_matrix(df2[abc], grid=False, figsize=(6, 6), diagonal=
↪'hist',
   ....:     hist_kwds={'density': True, 'bins': 25, 'lw': .25, 'ec': 'w'}
↪,
   ....:     s=1, marker='.');
   ....:
```



The sample uses a different random state and produces a different draw. Comparing `qdp` output is one way to see if 10000 simulations is adequate. In this case there is good agreement.

```
In [30]: qdp(df2)
Out[30]:
                   A            B            C         total
count   10000.000000 10000.000000 10000.000000  10000.000000
mean        9.991469    14.969470     4.987341     29.948280
```

(continues on next page)

(continued from previous page)

```
std        1.997495      7.389115      3.910557     10.902512
min        4.070312      1.984375      0.304688      8.242188
25%        8.570312      9.742188      2.414062     22.304688
50%        9.789062     13.406250      3.906250     27.867188
75%       11.179688     18.351562      6.273438     35.460938
max       19.398438     79.468750     45.937500    111.164062
cv         0.199920      0.493612      0.784097      0.364045
```

2.12.4 Applying the Iman-Conover Algorithm

The method `sample()` automatically applies the Iman-Conover algorithm (described in ../5_technical_guides/5_x_iman_conover). It is also easy to apply Iman-Conover to a dataframe using the method *`aggregate.utilities.iman_conover()`*. It reorders the input dataframe to have the same rank correlation as a multivariate normal reference sample with the desired linear correlation. Optionally, a multivariate t-distribution can be used as the reference.

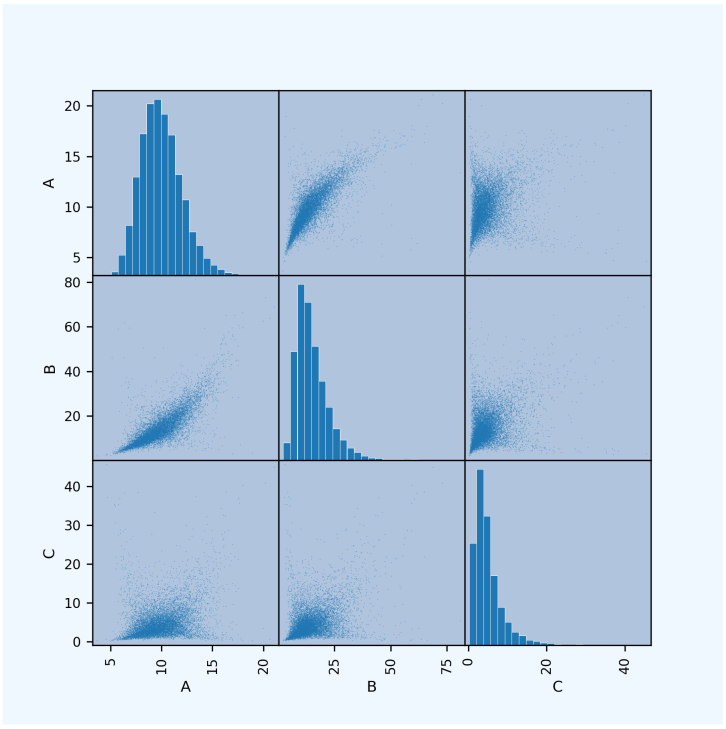
Examples.

Apply Iman-Conover to the sample `df` with target the correlation `rcm`, reusing the variables created in the previous section. The achieved correlation is close to that requested, as shown in the last two blocks.

```
In [31]: from aggregate import iman_conover

In [32]: import pandas as pd

In [33]: ans = iman_conover(df[abc], rcm, add_total=False)

In [34]: qd(pd.DataFrame(rcm, index=abc, columns=abc))

        A       B       C
A       1 0.80331 0.24378
B 0.80331       1 0.35449
C 0.24378 0.35449       1

In [35]: qd(ans.corr())

        A       B       C
A       1 0.78111 0.22247
B 0.78111       1 0.31237
C 0.22247 0.31237       1
```

Setting the argument `dof` uses a t-copula reference with `dof` degrees of freedom. The t-copula with low degrees of freedom can produce pinched multivariate distributions. Use with caution.

```
In [36]: ans = iman_conover(df[abc], rcm, dof=2, add_total=False)

In [37]: qd(ans.corr())

        A       B       C
A       1 0.76907 0.28673
B 0.76907       1 0.36892
C 0.28673 0.36892       1

In [38]: scatter_matrix(ans, grid=False, figsize=(6, 6), diagonal='hist',
   ....:     hist_kwds={'density': True, 'bins': 25, 'lw': .25, 'ec': 'w'},
   ....:     s=1, marker='.');
   ....:
```



See WP REF for ways to apply Iman-Conover with different reference distributions.

Details. Creating the independent scores for Iman-Conover is quite time consuming. They are cached for a given sample size. Second and subsequent calls are far quicker (an order of magnitude) than the first call.

2.12.5 Applying the Re-Arrangement Algorithm

The method `rearrangement_algorithm_max_VaR()` implements the re-arrangement algorithm described in ../5_technical_guides/5_x_rearrangement_algorithm. It returns only the tail of the re-arrangement, since values below the requested percentile are irrelevant.

Apply to `df` and request 0.999-VaR. The marginals are the 10 largest values. The algorithm permutes them to balance large and small observations.

```
In [39]: from aggregate import rearrangement_algorithm_max_VaR

In [40]: ans = rearrangement_algorithm_max_VaR(df.iloc[:, :3], .999)

In [41]: qd(ans, float_format=fc)

         A        B        C     total
0    21.08    63.78    38.59    123.45
6    20.22    65.18    38.40    123.80
1    20.63    62.50    40.80    123.93
3    18.93    65.02    40.24    124.20
5    17.98    60.66    45.56    124.20
2    18.02    68.88    37.33    124.23
7    17.75    61.75    44.91    124.41
9    19.99    66.28    38.22    124.49
8    17.69    73.03    37.12    127.84
4    17.59    81.07    37.03    135.70
```

Here are the stand-alone `sa` VaRs by marginal, in total for `df`, in total for the correlated `df2`, and the re-arrangement solutions `ra` for a range of different percentiles. The column `comon total` shows VaR for the comonotonic sum of the marginals (which equals the largest TVaR and variance re-arrangement).

```
In [42]: ps = [9000, 9500, 9900, 9960, 9990, 9999]

In [43]: sa = pd.concat([df[c].sort_values().reset_index(drop=True).iloc[ps] for c␣
↪in df]
   ....:                    +[df2.rename(columns={'total':'corr total'})['corr total
↪'].\
   ....:                      sort_values().reset_index(drop=True).iloc[ps]], axis=1)
   ....:

In [44]: sa['comon total'] = sa[abc].sum(1)

In [45]: ra = pd.concat([rearrangement_algorithm_max_VaR(df.iloc[:, :3], p/10000).
↪iloc[0]  for p in ps],
   ....:            axis=1, keys=ps).T
   ....:

In [46]: exhibit = pd.concat([sa, ra], axis=1, keys=['stand-alone', 're-arrangement
↪'])

In [47]: exhibit.index = [f'{x/10000:.2%}' for x in exhibit.index]

In [48]: exhibit.index.name = 'percentile'

In [49]: qd(exhibit, float_format=fc)

           stand-alone                                                     re-
↪arrangement           \
                     A        B         C    total  corr total  comon total          ␣
↪   A        B
percentile                                                                          ␣
↪
90.00%           12.59    24.75      9.73    41.48       44.22        47.06         ␣
↪14.66    29.73
95.00%           13.55    29.21     12.68    46.48       50.43        55.44         ␣
↪16.42    32.19
99.00%           15.45    39.98     20.55    58.15       65.26        75.98         ␣
↪17.75    44.51
99.60%           16.36    47.62     26.58    68.00       73.50        90.56         ␣
↪17.59    59.56
99.90%           17.59    60.66     37.03    80.20       93.51       115.28         ␣
↪21.08    63.78
99.99%           21.08    81.07     45.56   115.28      111.16       147.71         ␣
↪21.08    81.07


                  C    total
percentile
90.00%        12.62    57.02
95.00%        16.98    65.59
99.00%        26.37    88.62
99.60%        28.77   105.92
99.90%        38.59   123.45
99.99%        45.56   147.71
```

See also *Worked Example*.

2.12.6 Creating a Portfolio From a Sample

A Portfolio can be created from an existing sample by passing in a dataframe rather than a list of aggregates. This approach is useful when another model has created the sample, but the user wants to access other aggregate functionality. Each marginal in the sample is created as a dsev with the sampled outcomes. The p_total column used to set scenario probabilities if its is input, otherwise each scenario is treated as equally likely. The Portfolio ignores any the correlation structure of the sample; the marginals are treated as independent, but see *Using Samples and the Switcheroo Trick* for a way around this assumption.

Example.

Create a simple discrete sample from a three unit portfolio.

```
In [50]: sample = pd.DataFrame(
   ....:     {'A': [20, 22, 24, 6, 5, 6, 7, 8, 21, 3],
   ....:      'B': [20, 18, 16, 14, 12, 10, 8, 6, 4, 2],
   ....:      'C': [0, 0, 0, 0, 0, 0, 0, 0, 20, 40]})
   ....:

In [51]: qd(sample)

    A   B   C
0  20  20   0
1  22  18   0
2  24  16   0
3   6  14   0
4   5  12   0
5   6  10   0
6   7   8   0
7   8   6   0
8  21   4  20
9   3   2  40
```

Pass to Portfolio to create with these marginals. In this case, treat the marginals as discrete and update with bs=1.

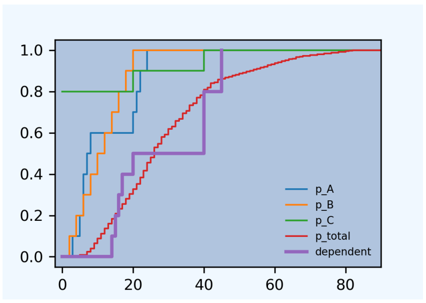
```
In [52]: from aggregate import Portfolio

In [53]: p03 = Portfolio('Samp:03', sample)

In [54]: p03.update(bs=1, log2=8)

In [55]: qd(p03)

             E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
unit  X
A     Freq      1                            0
      Sev    12.2     12.2           0 0.65142   0.65142 0.37792     0.37792
      Agg    12.2     12.2           0 0.65142   0.65142 0.37792     0.37792
B     Freq      1                            0
      Sev      11       11           0 0.52223   0.52223       0           0
      Agg      11       11           0 0.52223   0.52223       0           0
C     Freq      1                            0
      Sev       6        6 -3.3307e-16  2.1344    2.1344  1.9198      1.9198
      Agg       6        6 -3.3307e-16  2.1344    2.1344  1.9198      1.9198
total Freq      3                            0
      Sev  9.7333   9.7333           0 0.99572            1.0775
      Agg    29.2     29.2 -3.3307e-16 0.55238   0.55238  1.0061      1.0061
log2 = 8, bandwidth = 1, validation: not unreasonable.
```

The univariate statistics for each marginal are the same as the sample input, but because they added independently, the totals differ. The sample has negative correlation and a lower CV.

```
In [56]: sample['total'] = sample.sum(1)

In [57]: qdp(sample)
Out[57]:
               A          B          C      total
count  10.000000  10.000000  10.000000  10.000000
mean   12.200000  11.000000   6.000000  29.200000
std     7.947327   5.744563  12.806248  12.998461
min     3.000000   2.000000   0.000000  14.000000
25%     6.000000   6.500000   0.000000  16.250000
50%     7.500000  11.000000   0.000000  30.000000
75%    20.750000  15.500000   0.000000  40.000000
max    24.000000  20.000000  40.000000  45.000000
cv      0.651420   0.522233   2.134375   0.445153
```

The `Portfolio` total is a convolution of the input marginals and includes all possible combinations added independently. The figure plots the distribution functions.

```
In [58]: ax = p03.density_df.filter(regex='p_[ABCt]').cumsum().plot(
   ....:     drawstyle='steps-post', lw=1, figsize=(3.5, 2.45))
   ....:

In [59]: ax.plot(np.hstack((0, sample.total.sort_values())), np.linspace(0, 1, 11),
   ....:     drawstyle='steps-post', lw=2, label='dependent');
   ....:

In [60]: ax.set(xlim=[-2, 90]);

In [61]: ax.legend(loc='lower right');
```



2.12.7 Using Samples and the Switcheroo Trick

`Portfolio` objects created from a sample ignore the dependency structure; the `aggregate` convolution algorithm always assumes independence. It is highly desirable to retain the sample's dependency structure. Many calculations rely only on $\mathsf{E}[X_i \mid X]$ and not the input densities per se. Thus, we reflect dependency if we alter the values $\mathsf{E}[X_i \mid X]$ based on a sample and recompute everything that depends on them. The method `Portfolio.add_exa_sample()` implements this idea.

Example.

`sample` was chosen to have lots of ties - different ways of obtaining the same total outcome.

```
In [62]: qd(sample)

    A   B   C  total
0  20  20   0     40
1  22  18   0     40
2  24  16   0     40
3   6  14   0     20
4   5  12   0     17
```

(continues on next page)

(continued from previous page)

```
5   6  10   0      16
6   7   8   0      15
7   8   6   0      14
8  21   4  20      45
9   3   2  40      45
```

Apply `add_exa_sample` to the `sample` dataframe and look at the outcomes with positive probability. When a total outcome can occur in multiple ways, `exeqa_i` gives the average value of unit `i`. The function is applied to a copy of the original `Portfolio` object because it invalidates various internal states. The output dataframe is indexed by total loss. Notice that rows sum to the correct total.

```
In [63]: p03sw = Portfolio('Samp:03sw', sample)

In [64]: p03sw.update(bs=1, log2=8)

In [65]: df = p03sw.add_exa_sample(sample)

In [66]: qd(df.query('p_total > 0').filter(regex='p_total|exeqa_[ABC]'))

      p_total  exeqa_A  exeqa_B  exeqa_C
14.0      0.1        8        6        0
15.0      0.1        7        8        0
16.0      0.1        6       10        0
17.0      0.1        5       12        0
20.0      0.1        6       14        0
40.0      0.3       22       18        0
45.0      0.2       12        3       30
```

Swap the `density_df` dataframe — the **switcheroo trick**.

```
In [67]: p03sw.density_df = df
```

See the function `Portfolio.create_from_sample` for a single step create from sample, update, add exa calc, and switcheroo.

Most `Portfolio` spectral functions depend only on marginal conditional expectations. Applying these functions through `p03sw` reflects dependencies. Calibrate some distortions to a 15% return. The maximum loss is only 45, so use a 1-VaR, no default capital standard.

```
In [68]: p03sw.calibrate_distortions(ROEs=[0.15], Ps=[1], strict='ordered');

In [69]: qd(p03sw.distortion_df)

                          S    L       P      PQ       Q   COC    param       error
a    LR        method
45.0 0.934075 ccoc        0  29.2  31.261  2.2753  13.739  0.15    0.15            0
              ph          0  29.2  31.261  2.2753  13.739  0.15 0.80913   4.1444e-09
              wang        0  29.2  31.261  2.2753  13.739  0.15 0.18223  -1.6102e-07
              dual        0  29.2  31.261  2.2753  13.739  0.15   1.228  -9.8354e-08
              tvar        0  29.2  31.261  2.2753  13.739  0.15 0.12059    5.935e-06
```

Apply the PH and dual to the independent and dependent portfolios. Asset level 45 is the 0.861 percentile of the independent.

```
In [70]: d1 = p03sw.dists['ph']; d2 = p03sw.dists['dual']

In [71]: for d in [d1, d2]:
   ....:     print(d.name)
   ....:     print('='*74)
   ....:     pr = p03.price(1, d)
   ....:     pr45 = p03.price(.861, d)
```

(continues on next page)

(continued from previous page)

```
   ....:     prsw = p03sw.price(1, d)
   ....:     a = pd.concat((pr.df, pr45.df, prsw.df), keys=['pr', 'pr45', 'prsw'])
   ....:     qd(a, float_format=lambda x: f'{x:7.3f}')
   ....:
ph
==========================================================================

statistic                                  L       P       M       Q       a      ␣
↪LR      PQ  \
     distortion                 unit                                               ␣
↪
pr   Proportional Hazard, 0.809 A      12.200  12.944   0.744  13.129  26.073   0.
↪943   0.986
                                B      11.000  11.376   0.376   9.544  20.919   0.
↪967   1.192
                                C       6.000   8.400   2.400  28.608  37.008   0.
↪714   0.294
                                total  29.200  32.719   3.519  51.281  84.000   0.
↪892   0.638
pr45 Proportional Hazard, 0.809 A      11.675  12.059   0.384   3.624  15.682   0.
↪968   3.328
                                B      10.578  10.674   0.096   2.160  12.834   0.
↪991   4.941
                                C       4.803   6.457   1.654  10.027  16.484   0.
↪744   0.644
                                total  27.055  29.189   2.134  15.811  45.000   0.
↪927   1.846
prsw Proportional Hazard, 0.809 A      12.200  12.571   0.371   3.371  15.942   0.
↪970   3.729
                                B      11.000  10.532  -0.468  -0.187  10.345   1.
↪044 -56.386
                                C       6.000   8.158   2.158  10.555  18.713   0.
↪736   0.773
                                total  29.200  31.261   2.061  13.739  45.000   0.
↪934   2.275

statistic                                COC
     distortion                 unit
pr   Proportional Hazard, 0.809 A       0.057
                                B       0.039
                                C       0.084
                                total   0.069
pr45 Proportional Hazard, 0.809 A       0.106
                                B       0.045
                                C       0.165
                                total   0.135
prsw Proportional Hazard, 0.809 A       0.110
                                B       2.507
                                C       0.204
                                total   0.150
dual
==========================================================================

statistic                          L       P       M       Q       a      LR      ␣
↪PQ     COC
     distortion         unit                                                      ␣
↪
pr   Dual Moment, 1.228 A      12.200  13.061   0.861  15.011  28.071   0.934   0.
↪870   0.057
                        B      11.000  11.523   0.523  11.590  23.113   0.955   0.
↪994   0.045
```

(continues on next page)

(continued from previous page)

```
                              C       6.000   7.171   1.171  25.645  32.816   0.837   0.
↪280   0.046
                              total  29.200  31.754   2.554  52.246  84.000   0.920   0.
↪608   0.049
pr45 Dual Moment, 1.228 A      11.675  12.422   0.747   5.480  17.902   0.940   2.
↪267   0.136
                              B      10.578  11.009   0.432   4.186  15.195   0.961   2.
↪630   0.103
                              C       4.803   5.716   0.914   6.187  11.903   0.840   0.
↪924   0.148
                              total  27.055  29.148   2.093  15.852  45.000   0.928   1.
↪839   0.132
prsw Dual Moment, 1.228 A      12.200  12.874   0.674   4.947  17.821   0.948   2.
↪602   0.136
                              B      11.000  11.197   0.197   2.644  13.840   0.982   4.
↪235   0.074
                              C       6.000   7.191   1.191   6.148  13.339   0.834   1.
↪170   0.194
                              total  29.200  31.261   2.061  13.739  45.000   0.934   2.
↪275   0.150
```

2.12.8 Summary of Objects Created by DecL

Objects created by `build()` in this guide. Objects created directly by class constructors are not entered into the knowledge database.

```
In [72]: from aggregate import pprint_ex

In [73]: for n, r in build.qlist('^Samp:').iterrows():
   ....:     pprint_ex(r.program, split=20)
   ....:
```

2.13 Published Problems and Examples

Objectives: `aggregate` solutions to a wide selection of problems and examples from books (Loss Models, Loss Data Analytics), actuarial exam study notes, and academic papers. Demonstrates the method of solution and verifies the correctness of `aggregate` calculations.

Audience: Academics and researchers and actuarial students reading the cited works.

Prerequisites: `aggregate` programming; risk theory.

See also: Examples in *Reinsurance Pricing*.

Contents:

2.13.1 Grübel and Hermesmeier (1999)

Poisson/Levy Example

Here is an example from Grübel and Hermesmeier [1999]. The Levy distribution is a zero parameter distribution in `scipy.stats`. The paper considers an aggregate with Poisson(20) claim count. The Panjer recursion column can be replicated using more buckets and padding with `bs=1`. The function `exact` uses conditional probability to compute the aggregate probability of $x - 1/2 < X < x + 1/2$ exactly. The Levy is stable with index $\alpha = 1/2$, which means that

$$X_1 + \cdots + X_n =_d n^2 X$$

for iid Levy variables.

The other models use `log2=10`, no padding, and varying amounts of tilting.

```
In [1]: from aggregate import build, qd

In [2]: from scipy.stats import levy

In [3]: a = build('agg L 20 claim sev levy poisson', update=False)

In [4]: bs = 1

In [5]: a.update(log2=16, bs=bs, padding=2, normalize=False, tilt_vector=None)

In [6]: df = a.density_df.loc[[1, 10, 100, 1000], ['p_total']] / a.bs

In [7]: df.columns = ['accurate']

In [8]: def exact(x):
   ...:     lam = 20
   ...:     n = 100
   ...:     p = np.zeros(n)
   ...:     a = np.zeros(n)
   ...:     p[0] = np.exp(-lam)
   ...:     fz = levy()
   ...:     for i in range(1, n):
   ...:         p[i] = p[i-1] * lam / i
   ...:         a[i] = fz.cdf((x+0.5)/i**2) - fz.cdf((x-0.5)/i**2)
   ...:     return np.sum(p * a)
   ...:

In [9]: df['exact'] = [exact(i) for i in df.index]

# other models
In [10]: log2 = 10

In [11]: for tilt in [None, 1/1024, 5/1024, 25/1024]:
   ....:     a.update(log2=log2, bs=bs, padding=0, normalize=False, tilt_
↪vector=tilt)
   ....:     if tilt is None:
   ....:         tilt = 0
   ....:     df[f'tilt {tilt:.4f}'] = a.density_df.loc[[1, 10, 100, 1000], ['p_
↪total']]/a.bs
   ....:

In [12]: qd(df.iloc[:, [1,0,2,3,4, 5]], accuracy=3)

            exact   accurate  tilt 0.0000  tilt 0.0010  tilt 0.0049  tilt 0.0244
loss
1.0     1.0778e-07 2.4616e-07   0.00020645   7.3458e-05   1.5602e-06   2.4616e-07
10.0    3.0754e-05 3.4324e-05   0.00023799   0.00010668   3.5624e-05   3.4324e-05
100.0    0.0011555  0.0011559    0.0013212    0.0012148    0.0011569    0.0011559
1000.0 0.00020129 0.00020121   0.00021341   0.00020561   0.00020129   0.00020121
```

This table is identical to the table shown in the paper.

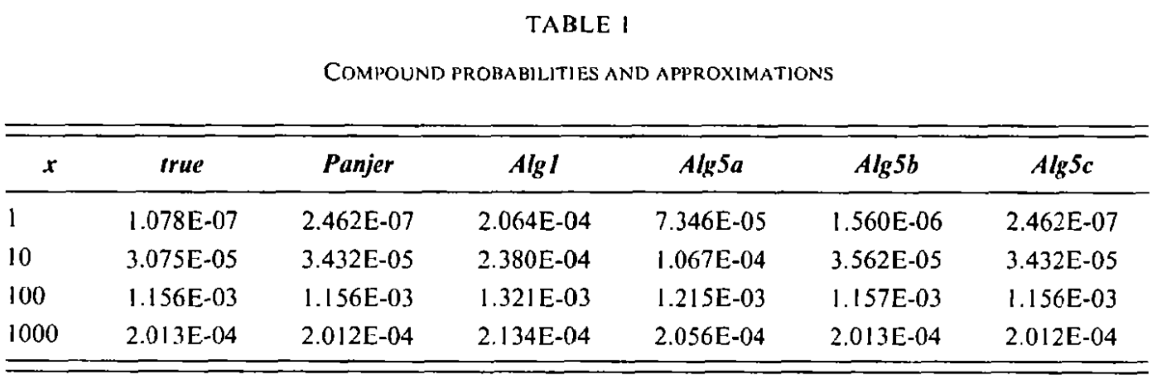


TABLE 1

COMPOUND PROBABILITIES AND APPROXIMATIONS

x	*true*	*Panjer*	*Alg1*	*Alg5a*	*Alg5b*	*Alg5c*
1	1.078E-07	2.462E-07	2.064E-04	7.346E-05	1.560E-06	2.462E-07
10	3.075E-05	3.432E-05	2.380E-04	1.067E-04	3.562E-05	3.432E-05
100	1.156E-03	1.156E-03	1.321E-03	1.215E-03	1.157E-03	1.156E-03
1000	2.013E-04	2.012E-04	2.134E-04	2.056E-04	2.013E-04	2.012E-04

2.13.2 Embrechts and Frei (2009)

Poisson/Pareto Example

Embrechts and Frei [2009], Panjer recursion versus FFT for compound distributions.

Consider a $\mathrm{Po}(\lambda) \vee \mathrm{Pareto}(\alpha, \beta)$, α is shape and β is scale.

```
In [1]: from aggregate import build, qd

In [2]: from pandas import option_context

In [3]: import matplotlib.pyplot as plt

In [4]: alpha = 4

In [5]: beta = 3

In [6]: freq = 20

In [7]: a = build(f'agg EF.1 {freq} claims '
   ...:           f'sev {beta} * pareto {alpha} - {beta} '
   ...:            'poisson', bs=1/8, log2=8, padding=0, normalize=False)
   ...:

In [8]: qd(a)

      E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq    20                       0.22361             0.22361
Sev      1   0.9985 -0.0014965  1.4142     1.3956   7.0711      5.1093
Agg     20   17.704   -0.11481  0.3873    0.38559   1.1619    -0.21402
log2 = 8, bandwidth = 1/8, validation: fails sev mean, agg mean, agg mean error >>␣
↪sev, possible aliasing; try larger bs.
```

The last dataframe shows poor accuracy. Try different ways to compute the aggregate: padding, tilting, and more buckets.

```
In [9]: from pandas import option_context

In [10]: df = a.density_df[['p_total']].rename(columns={'p_total': 'Pad 0, tilt 0'}
↪)

In [11]: a.update(bs=1/8, log2=8, padding=1, normalize=False)

In [12]: df['Pad 1, tilt 0'] = a.density_df.p_total
```

(continues on next page)

(continued from previous page)

```
In [13]: a.update(bs=1/8, log2=8, padding=2, normalize=False)

In [14]: df['Pad 2, tilt 0'] = a.density_df.p_total

In [15]: a.update(bs=1/8, log2=8, padding=0, tilt_vector=0.01, normalize=False)

In [16]: df['Pad 0, tilt 0.01'] = a.density_df.p_total

In [17]: a.update(bs=1/32, log2=16, padding=1, normalize=False)

In [18]: bit = a.density_df[['p_total']].rename(columns={'p_total': 'log2 16, pad␣
↪1, tilt 0'})

In [19]: qd(a)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    20                      0.22361            0.22361
Sev      1  0.99995 -5.4251e-05  1.4142    1.4144   7.0711      7.0284
Agg     20   19.999 -5.4252e-05  0.3873   0.38732   1.1619      1.1567
log2 = 16, bandwidth = 1/32, validation: not unreasonable.
```

The last dataframe shows a good approximation.

The next figure (compare Figure 1 in the paper, shown below) shows that padding, as recommended in Wang [1998], removes aliasing as effectively as padding, albeit at the expense of a longer FFT computation. The log density shows the aliasing is completely removed.

```
In [20]: f, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True, squeeze=True)

In [21]: ax0, ax1 = axs.flat

In [22]: df.plot(ax=ax0, logy=False)
Out[22]: <Axes: xlabel='loss'>

In [23]: df.plot(ax=ax1, logy=True)
Out[23]: <Axes: xlabel='loss'>

In [24]: ax0.legend(loc='upper left')
Out[24]: <matplotlib.legend.Legend at 0x7f808bd57ca0>

In [25]: ax1.legend(loc='lower right');
```

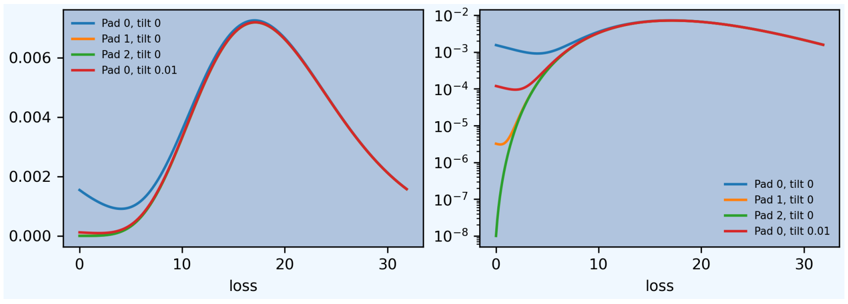
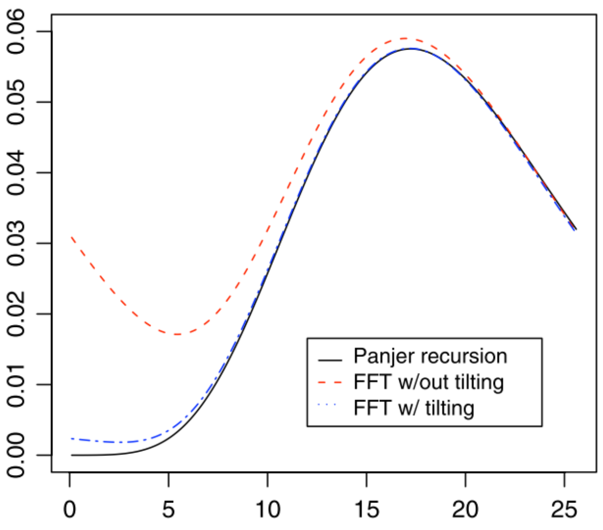
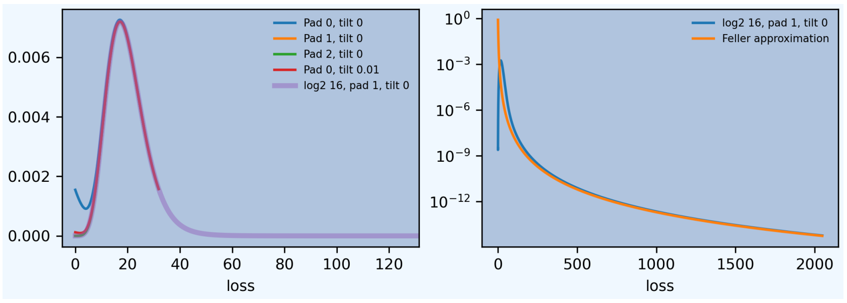


Fig. 1 Approximate densities for $Pois(20) \vee Pareto(4, 3)$



Clearly there is not enough *space* with only 2**8 buckets. Expanding to 2**16 and using a finer bucket covers a more realistic range. The log density plot shows a change in regime from Poisson body to Pareto tail. The extreme tail can be approximated by differentiating Feller's theorem, which says the survival function is converges to $20\mathsf{Pr}(X > x)$ where X is the Pareto severity (right hand plot). The multiplication by four accounts for the different `bs` values.

```
In [26]: f, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True, squeeze=True)

In [27]: ax0, ax1 = axs.flat

In [28]: df.plot(ax=ax0, logy=False)
Out[28]: <Axes: xlabel='loss'>

In [29]: (bit * 4).plot(ax=ax0, lw=3, alpha=.5);

In [30]: bit.plot(ax=ax1, logy=True);

# density from tail, need to divide by bs
In [31]: ax1.plot(bit.index, (20*4/3*a.bs)*(3/(3+bit.index))**5, label='Feller␣
↪approximation');

In [32]: ax0.set(xlim=[-5, a.q(0.99999)]);

In [33]: ax0.legend(loc='upper right');

In [34]: ax1.legend(loc='upper right');
```



Choice of Bandwidth (Bucket Size)

This example replicates parts of Table 1. As well as the 99.9%ile it shows the 99.9999%ile.

```
In [35]: import pandas as pd

In [36]: a = build('agg EF.2 50 claims sev expon poisson', update=False)

In [37]: ans = []

In [38]: for log2, bs in zip([10, 10, 10, 16, 16, 16, 16], [1, 1/2, 1/8, 1/8, 1/16,
↪ 1/64, 1/512]):
   ....:     a.update(log2=log2, bs=bs, padding=1)
   ....:     ans.append([log2, 1/bs, a.q(0.999), a.q(1-1e-6)])
   ....:

In [39]: df = pd.DataFrame(ans, columns=['log2', '1/bs', 'p999', 'p999999'])

In [40]: qd(df, accuracy=4)

   log2  1/bs   p999  p999999
0    10     1     84      107
1    10     2   84.5      108
2    10     8 85.125   108.25
3    16     8 85.125   108.25
4    16    16 85.125   108.25
5    16    64 85.109   108.23
6    16   512 85.105   108.23
```

2.13.3 Denuit (2019 and 2022)

Poisson/Discrete Example (6.1)

Example from Denuit [2019], Size-biased transform and conditional mean risk sharing, with application to p2p insurance and tontines.

```
In [1]: from aggregate import build, qd

In [2]: p = build('''
   ...: port Denuit6.1
   ...:     agg P1 0.08 claims dsev [1 2 3 4] [.1  .2  .4 .3] poisson
   ...:     agg P2 0.08 claims dsev [1 2 3 4] [.15 .25 .3 .3] poisson
   ...:     agg P3 0.10 claims dsev [1 2 3 4] [.1  .2  .4 .3] poisson
   ...:     agg P4 0.10 claims dsev [1 2 3 4] [.15 .25 .3 .3] poisson
   ...: ''', bs=1, log2=10)
   ...:

In [3]: qd(p)

            E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit  X
P1    Freq  0.08                       3.5355              3.5355
      Sev    2.9      2.9 -1.1102e-16 0.32531   0.32531 -0.51452    -0.51452
      Agg  0.232    0.232  4.0856e-13  3.7179    3.7179   3.9518      3.9518
P2    Freq  0.08                       3.5355              3.5355
      Sev   2.75     2.75           0 0.37921   0.37921 -0.28106    -0.28106
      Agg   0.22     0.22  3.7326e-13  3.7812    3.7812   4.0928      4.0928
P3    Freq   0.1                       3.1623              3.1623
      Sev    2.9      2.9 -1.1102e-16 0.32531   0.32531 -0.51452    -0.51452
      Agg   0.29     0.29 -4.5741e-14  3.3254    3.3254   3.5346      3.5346
```

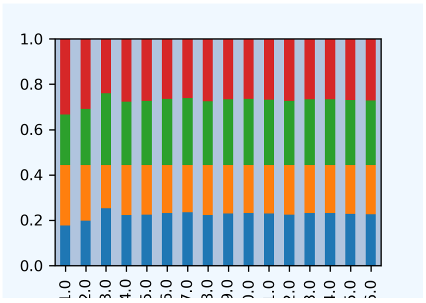
(continues on next page)

(continued from previous page)

```
P4     Freq   0.1                       3.1623             3.1623
       Sev   2.75     2.75            0 0.37921   0.37921 -0.28106    -0.28106
       Agg  0.275    0.275  1.1036e-13    3.382     3.382   3.6607      3.6607
total  Freq   0.36                      1.6667             1.6667
       Sev   2.825    2.825 -3.3307e-16 0.35299           -0.40097
       Agg  1.017    1.017 -5.6621e-15   1.7675    1.7675   1.8952      1.8952
log2 = 10, bandwidth = 1, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.
```

Computation of $\mathsf{E}[X_i \mid X = x]$ and $\mathsf{E}[X_i \mid X = x]/x$ as a function of x. The first function, called $\kappa_i(x)$ in PIR, is computed automatically by the `portfolio` class as `exeqa_line` (expectation given X equals x). The original figure is shown below.

```
In [4]: bit = p.density_df.query('p_total > 0').iloc[1:]

In [5]: rat = bit.filter(regex='exeqa_P').apply(
   ...:     lambda x: x / bit.loss.to_numpy(), axis=0)
   ...:

In [6]: ax = rat.plot.bar(ylim=[-0.05,1.05], stacked=True, figsize=(3.5, 2.45))

In [7]: ax.set(xlim=[-0.5, 15.5], ylim=[0,1]);

In [8]: ax.legend().set(visible=False);
```



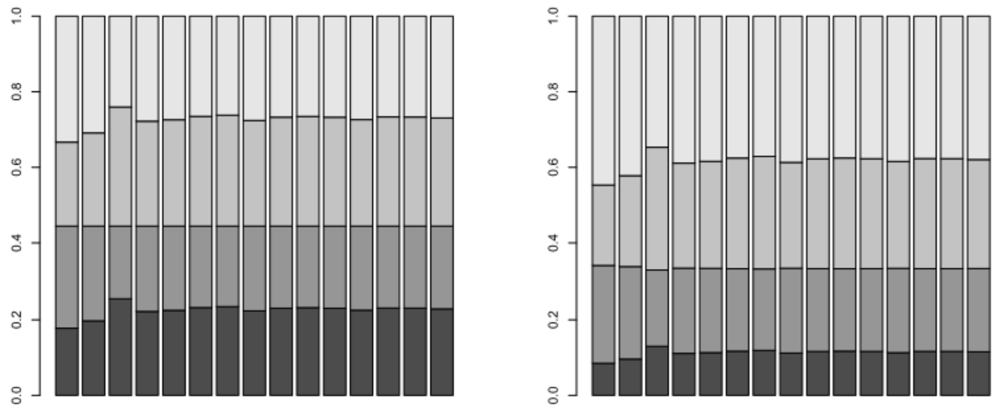


Figure 1: Respective shares $\mathrm{E}[X_i|S = s]/s$ of the total realized loss s contributed by each participant i in Case 1 (left panel) and Case 2 (right panel). From bottom to top, shares of participants 1 to 4.

All the values are available as a table. These are consistent with numbers mentioned in the text.

```
In [9]: from pandas import option_context

In [10]: b = bit.filter(regex='exeqa_P').apply(
   ....:     lambda x: x / bit.loss.to_numpy(), axis=0)
   ....:

In [11]: b.index = b.index.astype(int)

In [12]: b.index.name = 'a'

In [13]: qd(b)

    exeqa_P1  exeqa_P2  exeqa_P3  exeqa_P4
a
1    0.17778   0.26667   0.22222   0.33333
2    0.19729   0.24716   0.24661   0.30895
3    0.25219   0.19226   0.31523   0.24032
4    0.22212   0.22232   0.27765    0.2779
5    0.22524   0.21921   0.28155   0.27401
6    0.23238   0.21207   0.29047   0.26508
7    0.23486   0.20958   0.29358   0.26198
8    0.22365    0.2208   0.27956     0.276
9    0.23064    0.2138   0.28831   0.26725
10   0.23224    0.2122    0.2903   0.26526
11   0.23056   0.21388    0.2882   0.26735
12   0.22551   0.21893   0.28189   0.27366
..       ...       ...       ...       ...
30   0.22858   0.21586   0.28573   0.26982
31   0.22829   0.21615   0.28537   0.27017
32   0.22872   0.21571    0.2859   0.26963
33   0.22893   0.21549   0.28617   0.26937
34   0.22839     0.216   0.28551   0.26999
35   0.22829    0.2161   0.28537   0.27006
36   0.22865   0.21576   0.28577   0.26966
37   0.22863   0.21571   0.28579   0.26959
38   0.22816   0.21601   0.28522    0.2699
39   0.22816   0.21599   0.28513   0.26976
40   0.22849   0.21582   0.28545   0.26962
41   0.22806   0.21556   0.28498    0.2692
```

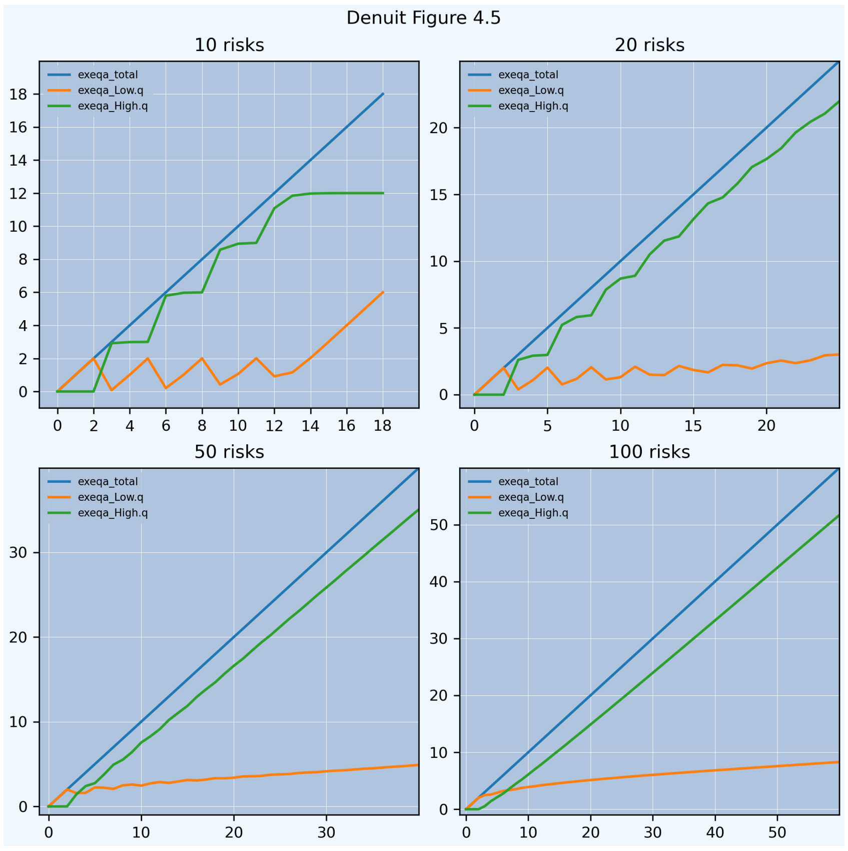
Proportion of expected loss by unit.

```
In [14]: bb = p.describe.xs('Agg', axis=0, level=1)[['E[X]']]

In [15]: qd(bb / bb.iloc[-1,0])

         E[X]
unit
P1    0.22812
P2    0.21632
P3    0.28515
P4     0.2704
total       1
```

Mortality Example and Figure

Example from Denuit *et al.* [2022], Mortality Credits with Large Survivor Funds. Reproducing Figure 4.5.

```
In [16]: import matplotlib.pyplot as plt; import pandas as pd

In [17]: wl = 0.6; wh = 1 - wl; ql = .1; qh = .2; al = 1; ah = 3

In [18]: ports = {}

In [19]: for n in (10, 20, 50, 100):
   ....:     ports[n] = build(f'port DR.4.3 '
   ....:           f'agg Low.q  {wl * n * ql} claims dsev [{al}] binomial {ql}'
   ....:           f'agg High.q {wh * n * qh} claims dsev [{ah}] binomial {qh}'
   ....:          , bs=1, log2=8)
   ....:

In [20]: audit = pd.concat([i.describe for i in ports.values()], keys=ports.keys(),
↪ names=['n', 'unit', 'X'])

In [21]: qd(audit.xs('Agg', axis=0, level=2)['E[X]'].unstack(1))

unit  Low.q  High.q  total
n
10      0.6     2.4      3
20      1.2     4.8      6
50        3      12     15
100       6      24     30

In [22]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 3.5), constrained_
↪layout=True, squeeze=True)

In [23]: for ax, (n, port), mx, t in zip(axs.flat, ports.items(), [20, 25, 40, 60],
↪ [2, 5, 10, 10]):
   ....:     lm = [-1, mx]
   ....:     port.density_df.query('p_total > 0').filter(regex='exeqa_[LHt]').
↪plot(ax=ax, xlim=lm, ylim=lm)
   ....:     ax.set_xticks(range(0, mx, t))
   ....:     ax.set_yticks(range(0, mx, t))
   ....:     ax.grid(lw=.25, c='w')
   ....:     ax.set(title=f'{n} risks')
   ....:

In [24]: fig.suptitle('Denuit Figure 4.5')
Out[24]: Text(0.5, 0.98, 'Denuit Figure 4.5')
```



2.13.4 Loss Data Analytics Book

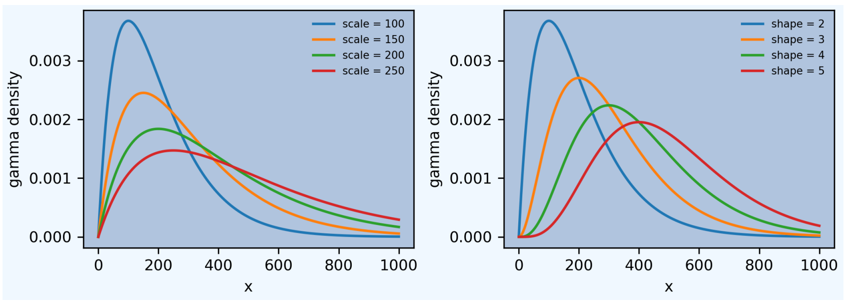
Examples from Jed Frees' open source actuarial software. Text and github source available on-line.

Contents

Distribution Examples

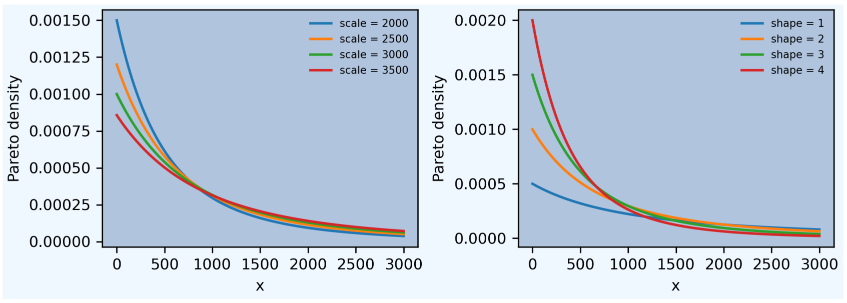
Gamma distribution

```
In [1]: import scipy.stats as ss

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt

In [4]: xs = np.linspace(0, 1000, 1001)

In [5]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True, squeeze=True)

In [6]: ax0, ax1 = axs.flat

In [7]: for scale in [100, 150, 200, 250]:
   ...:     ax0.plot(xs, ss.gamma(2, scale=scale).pdf(xs), label=f'scale = {scale}
↪')
   ...:

In [8]: for shape in [2, 3, 4, 5]:
   ...:     ax1.plot(xs, ss.gamma(shape, scale=100).pdf(xs), label=f'shape =
↪{shape}')
   ...:

In [9]: for ax in axs.flat:
   ...:     ax.legend(loc='upper right')
   ...:     ax.set(ylabel='gamma density', xlabel='x')
   ...:
```



Pareto distribution

```
In [10]: xs = np.linspace(0, 3000, 3001)

In [11]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True, squeeze=True)

In [12]: ax0, ax1 = axs.flat

In [13]: for scale in [2000, 2500, 3000, 3500]:
   ....:     ax0.plot(xs, ss.pareto(3, scale=scale, loc=-scale).pdf(xs), label=f
↪'scale = {scale}')
   ....:

In [14]: for shape in [1,2,3,4]:
   ....:     ax1.plot(xs, ss.pareto(shape, scale=2000, loc=-2000).pdf(xs), label=f
↪'shape = {shape}')
   ....:

In [15]: for ax in axs.flat:
   ....:     ax.legend(loc='upper right')
   ....:     ax.set(ylabel='Pareto density', xlabel='x')
   ....:
```



Weibull distribution

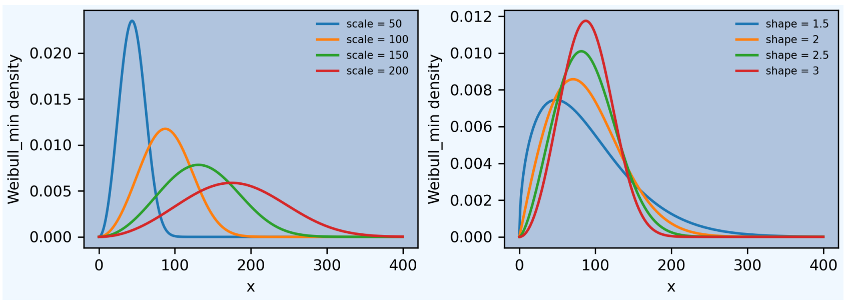
`scipy.stats` includes Weibull min (for positive x) and Weibull max (for negative x) distributions. We want the min version.

```
In [16]: xs = np.linspace(0, 400, 401)

In [17]: fig, axs = plt.subplots(1, 2, figsize=(2 * 3.5, 2.45), constrained_
↪layout=True, squeeze=True)

In [18]: ax0, ax1 = axs.flat

In [19]: for scale in [50, 100, 150, 200]:
   ....:     ax0.plot(xs, ss.weibull_min(3, scale=scale).pdf(xs), label=f'scale =
↪{scale}')
   ....:

In [20]: for shape in [1.5, 2, 2.5, 3]:
   ....:     ax1.plot(xs, ss.weibull_min(shape, scale=100).pdf(xs), label=f'shape␣
↪= {shape}')
   ....:

In [21]: for ax in axs.flat:
   ....:     ax.legend(loc='upper right')
   ....:     ax.set(ylabel='Weibull_min density', xlabel='x')
```

(continues on next page)

(continued from previous page)

```
   ....:
```



Mixture Example (3.3.5)

A collection of insurance policies consists of two types. 25% of policies are Type 1 and 75% of policies are Type 2. For a policy of Type 1, the loss amount per year follows an exponential distribution with mean 200, and for a policy of Type 2, the loss amount per year follows a Pareto distribution with parameters $\alpha = 3$ and $\theta = 200$. For a policy chosen at random from the entire collection of both types of policies, find the probability that the annual loss will be less than 100, and find the average loss.

Solution. The function `pmv` (print mean and variance) is a convenience.

```
In [22]: from aggregate import build, qd, mv

In [23]: def pmv(m, v):
   ....:     print(f'mean     = {m:.6g}\n'
   ....:           f'variance = {v:.7g}')
   ....:
```

Create the `Aggregate` object, display its `describe` dataframe and compare the cdf with the exact computation.

```
In [24]: a = build('agg lda.3.3.5 '
   ....:           '1 claim '
   ....:           'sev [200 200] * [expon pareto] [1 3] + [0 -200] wts [.25 .75] '
   ....:           'fixed',
   ....:           normalize=False)
   ....:

In [25]: qd(a)

      E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     1                           0
Sev    125      125 -2.3123e-05 1.4832    1.4754     inf      9.9477
Agg    125      125 -2.3123e-05 1.4832    1.4754     inf      9.9477
log2 = 16, bandwidth = 1/2, validation: fails sev cv, agg cv.

In [26]: a.sev.cdf(100), 0.25 * (1 - np.exp(-0.5)) + 0.75 * (1 - (2/3)**3)
Out[26]: (0.6261451128496194, 0.6261451128496194)
```

This example has a very thick tailed severity and it is best to specify `normalized=False` for the most accurate severity estimates. With default settings, `aggregate` suffers considerable discretization error, with an estimated mean well below the actual 125. The `sev.cdf` method exposes the actual underlying severity distribution cdf functions and reproduces the requested probability exactly. The object cdf function relies on the discretization and so is shifted by half a bucket size. (Also available: `sev.sf` and `sev.pdf`.)

```
In [27]: a.cdf(100), a.sev.cdf(100 + a.bs/2)
Out[27]: (0.626889166181137, 0.6268891661811365)
```

Coverage Modifications

Deductible Example (3.4.1)

A claim severity distribution is exponential with mean 1000. An insurance company will pay the amount of each claim in excess of a deductible of 100. Calculate the variance of the amount paid by the insurance company for one claim, including the possibility that the amount paid is 0.

Solution. In this case we must use unconditional severity to include the possibility that the amount paid is 0. This is done by adding `!` at the end of the severity specification. The moments are computed exactly without updating.

```
In [28]: import numpy as np

In [29]: a = build('agg lda.3.4.1 1 claim '
   ....:           'inf xs 100 sev 1000 * expon 1 ! '
   ....:           'fixed', update=False)
   ....:

In [30]: qd(a)

       E[X]  CV(X)  Skew(X)
X
Freq      1      0
Sev  904.84 1.1002  2.0257
Agg  904.84 1.1002  2.0257
log2 = 0, bandwidth = na, validation: n/a, not updated.

In [31]: m = 1000 * np.exp(-0.1)

In [32]: mv(a)
mean     = 904.837
variance = 990944.1
std dev  = 995.462

In [33]: pmv(m, (2 * 1000**2 * np.exp(-0.1)) - m**2)
mean     = 904.837
variance = 990944.1
```

Deductible Example (3.4.2)

For an insurance:

- Losses have a density function

$$f_X(x) = \begin{cases} 0.02x & 0 < x < 10, \\ 0 & \text{elsewhere.} \end{cases}$$

- The insurance has an ordinary deductible of 4 per loss.
- Y^P is the claim payment per payment random variable.

Solution. The trick here is to realize that X is a beta variable with $\alpha = 2$ and $\beta = 1$.

```
In [34]: a = build('agg lda.3.4.2 1 claim 6 xs 4 sev 10 * beta 2 1 fixed')

In [35]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq      1                            0
Sev  3.4286   3.4286 1.0348e-10 0.48947   0.48947 -0.29313    -0.29313
Agg  3.4286   3.4286 1.0348e-10 0.48947   0.48947 -0.29313    -0.29313
log2 = 16, bandwidth = 1/4096, validation: not unreasonable.
```

(continues on next page)

(continued from previous page)

```
In [36]: mv(a)
mean     = 3.42857
variance = 2.816327
std dev  = 1.67819
```

Limit Example (3.4.4)

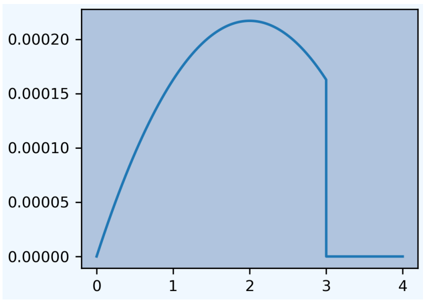
Under a group insurance policy, an insurer agrees to pay 100% of the medical bills incurred during the year by employees of a small company, up to a maximum total of one million dollars. The total amount of bills incurred, X, has *pdf*

$$f_X(x) = \begin{cases} \frac{x(4-x)}{9} & 0 < x < 3 \\ 0 & \text{elsewhere.} \end{cases}$$

where x is measured in millions. Calculate the total amount, in millions of dollars, the insurer would expect to pay under this policy.

Solution. In this case the distribution has no obvious parametric form—though it is related to a beta. We can solve it in `aggregate` by using a custom empirical distribution.

```
In [37]: xs = np.linspace(0, 4, 2**13, endpoint=False)

In [38]: F = np.where(xs<3,(xs * xs  * (2 - xs / 3)) / 9, 1)

In [39]: ps = np.diff(F, append=1)

In [40]: fig, ax = plt.subplots(1, 1, figsize=(3.5, 2.45), constrained_layout=True,
↪ squeeze=True)

In [41]: ax.plot(xs, ps);
```



When the empirical distribution has many entries it is faster to build the `Aggregate` object directly, rather than use DecL. The moments of the severity and aggregate distribution are computed from the numerical approximation during creation. There is no need to update the object.

```
In [42]: from aggregate import Aggregate

In [43]: a = Aggregate('Example', exp_en=1, sev_name='dhistogram', sev_xs=xs, sev_
↪ps=ps,
   ....:               exp_attachment=0, exp_limit=1, freq_name='fixed')
   ....:

In [44]: print(a)
aggregate object name    Example
claim count              1.00
frequency distribution   fixed
severity distribution    dhistogram, 1 xs 0.
        E[X]  CV(X)  Skew(X)
X
Freq        1      0      NaN
Sev   0.93514 0.1826  -2.9018
Agg   0.93514 0.1826  -2.9018
```

Limit and Deductible Example (3.4.5)

The ground up loss random variable for a health insurance policy in 2006 is modeled with X, a random variable with an exponential distribution having mean 1000. An insurance policy pays the loss above an ordinary deductible of 100, with a maximum annual payment of 500. The ground up loss random variable is expected to be 5% larger in 2007, but the insurance in 2007 has the same deductible and maximum payment as in 2006. Find the percentage increase in the expected cost per payment from 2006 to 2007.

Solution. Trend increases the ground-up severity distribution but not the limit and attachment. The calculation is performed exactly on creation; again, there is no need to update the `Aggregate` object.

```
In [45]: import pandas as pd

In [46]: a06 = build('agg X06 1 claim 500 xs 100 sev 1000 * expon fixed',␣
↪update=False)

In [47]: a07 = build('agg X07 1 claim 500 xs 100 sev 1050 * expon fixed',␣
↪update=False)

In [48]: ans = pd.concat((a06.describe, a07.describe), keys=['2006', '2007'])

In [49]: qd(ans)

             E[X]   CV(X)   Skew(X)
     X
2006 Freq       1       0       NaN
     Sev   393.47 0.40656   -1.1717
     Agg   393.47 0.40656   -1.1717
2007 Freq       1       0       NaN
     Sev    397.8 0.39691   -1.2342
     Agg    397.8 0.39691   -1.2342

In [50]: ans.iloc[5, 0] / ans.iloc[2, 0] - 1
Out[50]: 0.011000207063778467
```

Reinsurance Example (3.4.6, modified)

Losses arising in a certain portfolio have a two-parameter Pareto distribution with $\alpha = 5$ and $\theta = 3,600$. A reinsurance arrangement has been made, under which (a) the reinsurer accepts 15% of losses up to $u = 5,000$ and all amounts in excess of 5,000 and (b) the insurer pays for the remaining losses.

1. Express the random variables for the reinsurer's and the insurer's payments as a function of X, the portfolio losses.
2. Calculate the mean amount paid on a single claim by the insurer.
3. Calculate the standard deviation of the amount paid on a single claim by the insurer (retaining the 15% copay-ment).

Solution. The net position can be modeled as:

```
agg insurer.net 1 claim
sev 3600 * pareto 5 - 3600
occurrence net of 0.15 so 5000 xs 0 and inf xs 5000
fixed
```

but this involves the thick-tailed Pareto across its entire range. It is better to recognize the severity is limited by the second excess layer and proceed as follows.

```
In [51]: a = build('agg insurer.net 1 claim '
   ....:           '5000 xs 0 sev 3600 * pareto 5 - 3600 '
   ....:           'occurrence net of 0.15 so 5000 xs 0 '
   ....:           'fixed')
   ....:
```

(continues on next page)

(continued from previous page)

```
In [52]: qd(a)

       E[X] Est E[X] Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq      1                        0
Sev  872.37   741.53 -0.14998 1.1256    1.1256  2.0952       2.0952
Agg  872.37   741.53 -0.14998 1.1256    1.1256  2.0952       2.0952
log2 = 16, bandwidth = 1/4, validation: n/a, reinsurance.

In [53]: print('\n', a.agg_m, a.agg_sd)

 872.3650484486672 981.9314221673877
```

Aggregate Loss Distributions

Poisson/Discrete Example (5.3.1)

The number of accidents follows a Poisson distribution with mean 12. Each accident generates 1, 2, or 3 claimants with probabilities 1/2, 1/3, and 1/6 respectively.

Calculate the variance in the total number of claimants.

Solution.

```
In [54]: a = build('agg QU 12 claims dsev [1 2 3] [1/2 1/3 1/6] poisson')

In [55]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     12                     0.28868            0.28868
Sev  1.6667   1.6667 -1.1102e-16 0.44721   0.44721   0.6261       0.6261
Agg      20       20 -7.9936e-15 0.31623   0.31623  0.36366      0.36366
log2 = 10, bandwidth = 1, validation: not unreasonable.

In [56]: mv(a)
mean     = 20
variance = 40
std dev  = 6.32456
```

As always, a contains the (exact) full distribution of outcomes. We could answer any question about it.

Discrete Example (5.3.2)

You are the producer of a television quiz show that gives cash prizes. The number of prizes, N, and prize amount, X, have the following distributions:

n	$\Pr(N=n)$	x	$\Pr(X=x)$
1	0.8	0	0.2
2	0.2	100	0.7
		1000	0.1

Your budget for prizes equals the expected aggregate cash prizes plus the standard deviation of aggregate cash prizes. Calculate your budget.

Solution. Just a matter of translating into DecL. No need to update the object.

```
In [57]: a = build('agg lda.5.3.2 dfreq [1 2] [.8 .2] '
   ....:            'dsev [0 100 1000] [.2 .7 .1]', update=False)
   ....:
```

(continues on next page)

(continued from previous page)

```
In [58]: display(a)
lda.5.3.2, <aggregate.distributions.Aggregate object at 0x7f808c62a440>

In [59]: mv(a)
mean     = 204
variance = 98344
std dev  = 313.598

In [60]: a.agg_m + a.agg_sd
Out[60]: 517.5984693840198
```

Geometric/Discrete Example (5.3.3 and 5.4.1)

The number of claims in a period has a geometric distribution with mean 4. The amount of each claim X follows $\Pr(X = x) = 0.25,\ x = 1, 2, 3, 4$, i.e. a discrete uniform distribution on $\{1, 2, 3, 4\}$. The number of claims and the claim amounts are independent. Let S_N denote the aggregate claim amount in the period. Calculate $F_{S_N}(3)$.

Solution. We can compute the entire distribution. Here we show up to the 99th percentile. If the probability clause in `dsev` is omitted then all outcomes are treated as equally likely.

```
In [61]: a = build('agg lda.5.3.3 4 claims dsev [1:4] geometric')

In [62]: qd(a, accuracy=4)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     4                        1.118              2.0125
Sev    2.5      2.5           0 0.44721   0.44721        0           0
Agg     10       10 -1.9499e-10  1.1402    1.1402    2.024       2.024
log2 = 8, bandwidth = 1, validation: not unreasonable.

In [63]: b = a.density_df.loc[0:a.q(0.99), ['p_total', 'F']]

In [64]: b.index = b.index.astype(int)

In [65]: qd(b, accuracy=4)

        p_total       F
loss
0           0.2     0.2
1          0.04    0.24
2         0.048   0.288
3        0.0576  0.3456
4       0.06912 0.41472
5      0.042944 0.45766
6      0.043533  0.5012
7      0.042639 0.54384
8      0.039647 0.58348
9      0.033753 0.61724
10     0.031914 0.64915
11     0.029591 0.67874
...         ...     ...
40    0.0023287 0.97449
41    0.0021339 0.97662
42    0.0019554 0.97858
43    0.0017918 0.98037
44     0.001642 0.98201
45    0.0015046 0.98352
46    0.0013788  0.9849
47    0.0012634 0.98616
48    0.0011577 0.98732
49    0.0010609 0.98838
```

(continues on next page)

(continued from previous page)

```
50   0.00097217 0.98935
51   0.00089085 0.99024
```

Moments Example (5.3.4)

You are given:

	Mean	Standard Deviation
Number of Claims	8	3
Individual Losses	10, 000	3, 937

As a benchmark, use the normal approximation to determine the probability that the aggregate loss will exceed 150% of the expected loss.

Solution. Use the `MomentAggregator` class to compute the moments of an aggregate from those of frequency and severity.

```
In [66]: import scipy.stats as ss

In [67]: from aggregate import MomentAggregator

In [68]: mom = MomentAggregator.agg_from_fs2(8, 9, 10000, 3937**2)

In [69]: fz = ss.norm(loc=mom.ex, scale=mom.sd)

In [70]: mom['prob'] = fz.sf(1.5*mom.ex)

In [71]: qd(mom)

ex          80000
var     1.024e+09
sd          32000
cv            0.4
prob      0.10565
```

Poisson/Uniform Example (5.3.5 and 5.4.2)

For an individual over 65:

1. The number of pharmacy claims is a Poisson random variable with mean 25.
2. The amount of each pharmacy claim is uniformly distributed between 5 and 95.
3. The amounts of the claims and the number of claims are mutually independent.

Estimate the probability that aggregate claims for this individual will exceed 2000 using the normal approximation.

Solution. Here is a close-to exact solution in addition to the normal approximation. Note that the uniform distribution has no shape parameter. The severity is made by shifting and scaling the base. Scaling is like multiplication and is applied before the location (addition) shift.

```
In [72]: a = build('agg Pharma 25 claims sev 90 * uniform + 5 poisson')

In [73]: qd(a)

      E[X] Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq    25                            0.2                  0.2
Sev     50        50           0  0.51962    0.51962         0            0
Agg   1250      1250 2.6645e-15  0.22539    0.22539   0.25293      0.25293
log2 = 16, bandwidth = 1/16, validation: not unreasonable.
```

Here are the moments for the approximation. The `approximate` function returns a `scipy.stats` frozen normal object, which yields the approximation.

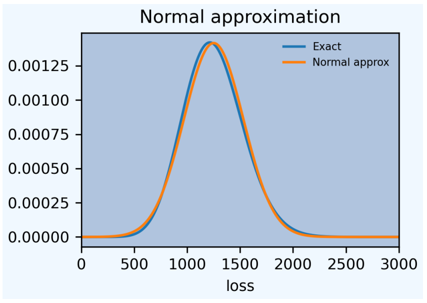
```
In [74]: print(a.sf(2000), a.agg_m, a.agg_var)
0.00698200305408303 1250.0 79375.0

In [75]: fz = a.approximate('norm')

In [76]: fz.sf(2000), a.sf(2000)
Out[76]: (0.0038830948902722645, 0.00698200305408303)
```

`approximate` will also provide (shifted) gamma and lognormal fits.

```
In [77]: approx = a.approximate('all')

In [78]: b = pd.DataFrame([[k, v.sf(2000)] for k, v in approx.items()],
   ....:                  columns=['approx', 'prob']).set_index('approx')
   ....:

In [79]: b.loc['exact'] = a.sf(2000)

In [80]: b.sort_values('prob')
Out[80]:
              prob
approx
norm      0.003883
exact     0.006982
sgamma    0.007101
slognorm  0.007175
gamma     0.009789
lognorm   0.013118
```

Here is a comparison of the FFT model with the normal approximation. Example 5.4.2 derives a similar probability using simulation.

```
In [81]: fig, ax = plt.subplots(1, 1, figsize=(3.5, 2.45), constrained_layout=True,
↪ squeeze=True)

In [82]: (a.density_df.p / a.bs).plot(label='Exact', ax=ax);

In [83]: ax.plot(a.xs, fz.pdf(a.xs), label='Normal approx');

In [84]: ax.set(xlim=[0, 3000], title='Normal approximation');

In [85]: ax.legend(loc='upper right');
```



Geometric/Discrete Example (5.3.6 and 5.3.7)

In a given week, the number of projects that require you to work overtime has a geometric distribution with $\beta = 2$. For each project, the distribution of the number of overtime hours in the week, X, is as follows:

x	$f(x)$
5	0.2
10	0.3
20	0.5

The number of projects and the number of overtime hours are independent. You will get paid for overtime hours in excess of 15 hours in the week. Calculate the expected number of overtime hours for which you will get paid in the week.

Solution. This is a one-liner in `aggregate`. Remember that aggregate reinsurance is specified after the frequency clause. The first column in the describe dataframe shows the analytic gross answer and the second the FFT-computed net.

```
In [86]: a = build('agg Projects 2 claims '
   ....:           'dsev [5 10 20] [.2 .3 .5] geometric '
   ....:           'aggregate net of 15 xs 0')
   ....:

In [87]: qd(a)

      E[X] Est E[X] Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq     2                     1.2247               2.0412
Sev     14       14         0 0.44607   0.44607 -0.23403    -0.23403
Agg     28   18.807 -0.32831   1.2647    1.6713   2.0725      2.6373
log2 = 9, bandwidth = 1, validation: n/a, reinsurance.
```

Example 5.3.7 uses a recursive calculation in steps of 5. We can replicate that using an aggregate tower. The `reinsurance_audit_df` provides ceded and net statistics by layer. Here we extract just the ceded part to get the excess (overtime).

```
In [88]: a1 = build('agg Projects.1 2 claims '
   ....:           'dsev [5 10 20] [.2 .3 .5] geometric '
   ....:           'aggregate net of tower [0 5 10 15 inf]')
   ....:

In [89]: b = a1.reinsurance_audit_df.xs('ceded', axis=1, level=0)

In [90]: b['cumul ex'] = b.ex[::-1].cumsum() - a.agg_m

In [91]: qd(b, accuracy=4)

                            ex     var      sd       cv      skew     cumul ex
kind share limit attach
agg  1.0    5.0   0.0    3.3333 5.5556   2.357 0.70711 -0.70711           28
                  5.0    3.1111 5.8765 2.4242 0.77919 -0.50418       24.666
                  10.0   2.7481 6.1884 2.4877 0.90521  -0.1995       21.555
            inf   15.0   18.807     988 31.433  1.6713   2.6373       18.807
     all    inf   gup        28 1253.9   35.41  1.2647   2.0713 -0.00026862
```

Zero-Modified Poisson/Burr Example (5.5.4)

Aggregate losses are modeled as follows:

1. The number of losses follows a zero-modified Poisson distribution with $\lambda = 3$ and $p_0^M = 0.5$.
2. The amount of each loss has a Burr distribution with $\alpha = 3, \theta = 50, \gamma = 1$.
3. There is a deductible of $d = 30$ on each loss.
4. The number of losses and the amounts of the losses are mutually independent.

Calculate $\mathsf{E}(N^P)$ and $\mathsf{Var}(N^P)$.

Solution.

Todo: Implement ZT and ZM!

Negative Binomial Example (5.5.5 and 5.5.6 modified)

A group dental policy has a negative binomial claim count distribution with mean 300 and variance 800. Ground-up severity is given by the following table:

Severity	Probability
40	0.25
80	0.25
120	0.25
200	0.25

You expect severity to increase 50% with no change in frequency. You decide to impose a per claim deductible of 100. Calculate the expected total claim payment S after these changes. What is the variance of the total claim payment, $\mathsf{Var}(S)$? (Modified:) Compare the aggregate distributions before and after the policy change.

Solution. A negative binomial with mean 300 and variance 800 has $8/3 = 1 + 300c$ and giving a mixing cv of $\sqrt{c} = (5/900)^{0.5} = 0.0745$. Hence the aggregate program is

TODO: why is the answer not exact?

```
In [92]: cv = ((8 / 3 - 1) / 300)**0.5

In [93]: a0 = build(f'agg Original 300 claims dsev [4 8 12 20] mixed gamma {cv}')

In [94]: a1 = build(f'agg Revised 300 claims inf xs 10 '
   ....:            f'sev dhistogram xps [{4*1.5} {8*1.5} {12*1.5} {20*1.5}] !␣
↪mixed gamma {cv}')
   ....:

In [95]: qd(a0)

      E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   300                      0.094281            0.15321
Sev     11       11           0  0.53783   0.53783  0.43465     0.43465
Agg   3300     3300 -3.3207e-13 0.099263  0.099263  0.15833     0.15832
log2 = 20, bandwidth = 1, validation: not unreasonable.

In [96]: qd(a1)

      E[X] Est E[X]   Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   300                     0.094281            0.15321
Sev    7.5      7.5 4.6567e-11   1.0392    1.0392   0.7207      0.7207
Agg   2250     2250 4.4878e-11  0.11175   0.11175  0.16722     0.16669
log2 = 20, bandwidth = 1, validation: not unreasonable.

In [97]: mv(a0)
mean     = 3300
variance = 107300
std dev  = 327.567

In [98]: mv(a1)
mean     = 2250
variance = 63225
std dev  = 251.446
```

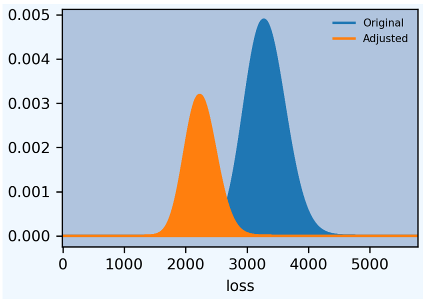
Here is a comparison of the two densities.

```
In [99]: fig, ax = plt.subplots(1, 1, figsize=(3.5, 2.45), constrained_layout=True,
↪ squeeze=True)

In [100]: a0.density_df.p_total.plot(ax=ax, label='Original');

In [101]: a1.density_df.p_total.plot(ax=ax, label='Adjusted');
```

(continues on next page)

(continued from previous page)

```
In [102]: ax.set(xlim=[-10, 1.25 * a0.q(0.9999)]);

In [103]: ax.legend(loc='upper right');
```



Poisson/Exponential Coverage and Underwriting Modification (Example 5.5.7)

A company insures a fleet of vehicles. Aggregate losses have a compound Poisson distribution. The expected number of losses is 20. Loss amounts, regardless of vehicle type, have exponential distribution with $\theta = 200$. To reduce the cost of the insurance, two modifications are to be made:

1. A certain type of vehicle will not be insured. It is estimated that this will reduce loss frequency by 20%.
2. A deductible of 100 per loss will be imposed.

Calculate the expected aggregate amount paid by the insurer after the modifications.

Solution. The `!` at the end of the severity clause indicates unconditional severity (including zero claims that fail to meet the deductible).

```
In [104]: a = build(f'agg Auto {20 * 0.8} claims inf xs 100 sev 200 * expon !␣
↪poisson')

In [105]: qd(a)

       E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
X
Freq     16                          0.25              0.25
Sev  121.31    121.31 -6.5104e-08  1.5157     1.5157   2.4172       2.4172
Agg  1940.9    1940.9 -6.5104e-08 0.45397    0.45397  0.68096      0.68096
log2 = 16, bandwidth = 1/4, validation: not unreasonable.
```

If severity is conditional there are 16 claims in excess of the deductible, giving a much higher number. The mean severity is still 200 because of the exponential's memoryless property.

```
In [106]: a = build(f'agg Auto {20 * 0.8} claims inf xs 100 sev 200 * expon poisson
↪')

In [107]: qd(a)

      E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
X
Freq    16                          0.25              0.25
Sev    200       200 -6.5104e-08       1          1        2            2
Agg   3200      3200 -6.5104e-08 0.35355    0.35355  0.53033      0.53033
log2 = 16, bandwidth = 1/4, validation: not unreasonable.
```

Portfolio Management

VaR for a Discrete Variable Example (10.3.4)

Consider an insurance loss random variable with the following probability distribution:

$$\Pr[X = x] = \begin{cases} 0.75, & \text{for } x = 1 \\ 0.20, & \text{for } x = 3 \\ 0.05, & \text{for } x = 4. \end{cases}$$

Determine the VaR at $q = 0.6, 0.9, 0.95, 0.95001$.

Solution.

```
In [108]: a = build('agg VaR 1 claim dsev [1 3 4] [.75 .2 .05] fixed')

In [109]: [a.q(i) for i in [.6, .9, .95, .9501]]
Out[109]: [1.0, 3.0, 3.0, 4.0]
```

Multi-Unit Telecom Management Example (10.4.3.3)

You are the Chief Risk Officer of a telecommunications firm. Your firm has several property and liability risks. We will consider:

- X_1, buildings, modeled using a gamma distribution with mean 200 and scale parameter 100.
- X_2, motor vehicles, modeled using a gamma distribution with mean 400 and scale parameter 200.
- X_3, directors and executive officers risk, modeled using a Pareto distribution with mean 1000 and scale parameter 1000.
- X_4, cyber risks, modeled using a Pareto distribution with mean 1000 and scale parameter 2000.

Denote the total risk as $X = X_1 + X_2 + X_3 + X_4$. For simplicity, you assume that these risks are independent. (Later, we will consider the more complex case of dependence.)

To manage the risk, you seek some insurance protection. You wish to manage internally small building and motor vehicles amounts, up to M_1 and M_2, respectively. You seek insurance to cover all other risks. Specifically, the insurer's portion is

$$Y_{insurer} = (X_1 - M_1)_+ + (X_2 - M_2)_+ + X_3 + X_4,$$

so that your retained risk is $Y_{retained} = X - Y_{insurer} = \min(X_1, M_1) + \min(X_2, M_2)$. Using deductibles $M_1 = 100$ and $M_2 = 200$:

1. Determine the expected claim amount of (i) that retained, (ii) that accepted by the insurer, and (iii) the total overall amount.
2. Determine the 80th, 90th, 95th, and 99th percentiles for (i) that retained, (ii) that accepted by the insurer, and (iii) the total overall amount.
3. Compare the distributions by plotting the densities for (i) that retained, (ii) that accepted by the insurer, and (iii) the total overall amount.

Solution. Begin by figuring the gamma and Pareto parameters. For a gamma, the mean equals shape times scale, so shape equals 2 for building and motor. For a Pareto, the mean equals scale / (shape - 1), so shape equals 2 (no variance) for D&O and 3 for cyber (no third moment). We model the results using three `Portfolio` objects, one for the retention, one for the insured amount, and one total. In each case the distribution gives total losses; the frequency component is trivial.

Since the insured and total aggregates have no variance it is hard to estimate an appropriate bucket size. The default method uses the standard deviation as a scale factor. We must use judgement (or trial and error), and select `log2=18` and `bs=1` to ensure there is enough "space". Checking the describe dataframe shows these values match the means well by unit and in total. The insured severity must be made unconditional.

```
In [110]: from aggregate import build

In [111]: retained = build('''port retained
   .....:     agg building 1 claim 100 xs 0 sev 100 * gamma 2 fixed
   .....:     agg motor    1 claim 200 xs 0 sev 200 * gamma 2 fixed
   .....: ''')
   .....:

In [112]: qd(retained)

                E[X] Est E[X]   Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
unit     X
building Freq      1                           0
         Sev  89.636   89.636 1.0437e-10 0.23985   0.23985 -2.1029     -2.1029
         Agg  89.636   89.636 1.0437e-10 0.23985   0.23985 -2.1029     -2.1029
motor    Freq      1                           0
         Sev  179.27   179.27 2.6093e-11 0.23985   0.23985 -2.1029     -2.1029
         Agg  179.27   179.27 2.6093e-11 0.23985   0.23985 -2.1029     -2.1029
total    Freq      2                           0
         Sev  134.45   134.45 5.2187e-11 0.41837           -0.0046
         Agg  268.91   268.91 5.2187e-11 0.17878   0.17878 -1.6928     -1.6928
log2 = 16, bandwidth = 1/128, validation: not unreasonable.

In [113]: insured = build('''port insured
   .....:     agg building 1 claim inf xs 100 sev 100 * gamma 2 ! fixed
   .....:     agg motor    1 claim inf xs 200 sev 200 * gamma 2 ! fixed
   .....:     agg d.and.o  1 claim            sev 1000 * pareto 2 - 1000 fixed
   .....:     agg cyber    1 claim            sev 2000 * pareto 3 - 2000 fixed
   .....: ''', log2=18, bs=1)
   .....:

In [114]: qd(insured)

                E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
unit     X
building Freq      1                            0
         Sev  110.36   110.36 -1.3889e-06 1.1901    1.1901   1.757       1.757
         Agg  110.36   110.36 -1.3889e-06 1.1901    1.1901   1.757       1.757
motor    Freq      1                            0
         Sev  220.73   220.73 -3.4722e-07 1.1901    1.1901   1.757       1.757
         Agg  220.73   220.73 -3.4722e-07 1.1901    1.1901   1.757       1.757
d.and.o  Freq      1                            0
         Sev    1000   992.43  -0.0075717    inf    2.6992              24.633
         Agg    1000   992.43  -0.0075717    inf    2.6992              24.633
cyber    Freq      1                            0
         Sev    1000   999.83 -0.00017075 1.7321    1.7062     inf      12.894
         Agg    1000   999.83 -0.00017075 1.7321    1.7062     inf      12.894
total    Freq      4                            0
         Sev  582.77   580.84  -0.0033215    inf
         Agg  2331.1   2323.3  -0.0033392    inf    1.3721              16.508
log2 = 18, bandwidth = 1, validation: fails sev mean, agg mean.

In [115]: total = build('''port total
   .....:     agg building 1 claim sev 100 * gamma 2 fixed
   .....:     agg motor    1 claim sev 200 * gamma 2 fixed
   .....:     agg d.and.o  1 claim sev 1000 * pareto 2 - 1000 fixed
   .....:     agg cyber    1 claim sev 2000 * pareto 3 - 2000 fixed
   .....: ''', log2=18, bs=1)
   .....:

In [116]: qd(total)
```

(continues on next page)

(continued from previous page)

```
                 E[X] Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
unit     X
building Freq     1                              0
         Sev    200      200 -1.2153e-11 0.70711   0.70711  1.4142      1.4142
         Agg    200      200 -1.2153e-11 0.70711   0.70711  1.4142      1.4142
motor    Freq     1                              0
         Sev    400      400 -7.5939e-13 0.70711   0.70711  1.4142      1.4142
         Agg    400      400 -7.5939e-13 0.70711   0.70711  1.4142      1.4142
d.and.o  Freq     1                              0
         Sev   1000   992.43  -0.0075717     inf    2.6992              24.633
         Agg   1000   992.43  -0.0075717     inf    2.6992              24.633
cyber    Freq     1                              0
         Sev   1000   999.83 -0.00017075  1.7321    1.7062     inf      12.894
         Agg   1000   999.83 -0.00017075  1.7321    1.7062     inf      12.894
total    Freq     4                              0
         Sev    650   648.06  -0.0029779     inf
         Agg   2600   2592.2  -0.0029969     inf    1.2304              16.463
log2 = 18, bandwidth = 1, validation: fails sev mean, agg mean.
```

The spacing in the agg programs is for clarity. We could also program using `dfreq` as `agg motor dfreq [1] 100 xs 0 sev...`. Next, assemble the requested data elements.

```
In [117]: pfs = [retained, insured, total]

In [118]: answers = pd.DataFrame(columns=['retained', 'insured', 'total'])

In [119]: answers.index.name = 'statistic'

In [120]: answers.loc['expected claim amount'] = [x.agg_m for x in pfs]

In [121]: for p in [.8, .9, .95, .99]:
   .....:     answers.loc[f'claim p_{p:.2f}'] = [x.q(p) for x in pfs]
   .....:

In [122]: qd(answers)

                       retained  insured  total
statistic
expected claim amount    268.91   2331.1   2600
claim p_0.80                300     3090   3364
claim p_0.90                300     4483   4756
claim p_0.95                300     6244   6516
claim p_0.99                300    12706  12977
```

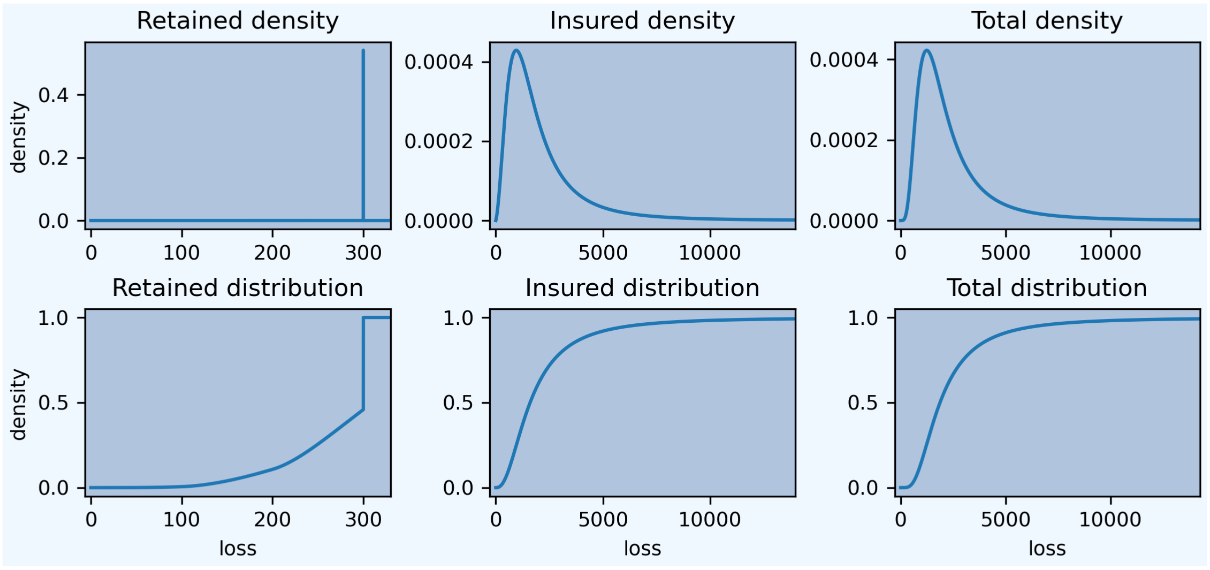
Finally, plot the densities. Compared to the text plot, the FFT reveals a discontinuous distribution for retained loss, with a large mass at 300. This is clearer on the lower plots, which show the distribution functions.

```
In [123]: fig, axs = plt.subplots(2, 3, figsize=(7.5, 3.5), constrained_
↪layout=True, squeeze=True)

In [124]: xl = {}

In [125]: for ax, pf in zip(axs.flat, pfs):
   .....:     pf.density_df.p_total.plot(ax=ax)
   .....:     q = pf.q(0.99) * 1.1
   .....:     xl[hash(pf)] = [-q / 50, q]
   .....:     ax.set(xlim=xl[hash(pf)], title=pf.name.title() + ' density')
   .....:     if pf is retained:
   .....:         ax.set(ylabel='density')
   .....:
```

(continues on next page)

(continued from previous page)

```
In [126]: for ax, pf in zip(axs.flat[3:], pfs):
   .....:     pf.density_df.F.plot(ax=ax)
   .....:     ax.set(xlim=xl[hash(pf)], title=pf.name.title() + ' distribution',␣
↪xlabel='loss')
   .....:     if pf is retained:
   .....:         ax.set(ylabel='density')
   .....:
```



2.13.5 Loss Models Book

Examples from the text Klugman *et al.* [2019], Loss Models: from data to decisions. The Loss models book is used as a text for several actuarial society exams and many college courses. KPW is shorthand for Loss Models.

Contents

- *Exercise 9.57 and 58*
- *Exercise 9.59*
- *Exercise 9.60*
- *Example 9.14*
- *Exercise 9.63*
- *Example 9.15 and 18*
- *Example 9.16 and 17*
- *Exercise 9.73*
- *Exercise 9.74*

Method of Moments Approximations, Examples 9.3 and 9.4

The observed mean (and standard deviation) of the number of claims and the individual losses over the past 10 months are 6.7 (2.3) and 179,247 (52,141), respectively. Determine the mean and standard deviation of aggregate claims per month.

```
In [1]: from aggregate import build, qd, mv, MomentAggregator, round_bucket

In [2]: import scipy.stats as ss

In [3]: import pandas as pd

In [4]: import numpy as np

In [5]: import matplotlib.pyplot as plt

In [6]: moms = MomentAggregator.agg_from_fs2(6.7, 2.3**2, 179247, 52141**2)

In [7]: moms
Out[7]:
ex      1.200955e+06
var     1.881802e+11
sd      4.337974e+05
cv      3.612104e-01
dtype: float64
```

Using normal and lognormal distributions as approximating distributions for aggregate claims, calculate the probability that claims will exceed 140% of expected costs.

```
In [8]: fzn = ss.norm(loc=moms.ex, scale=moms.sd)

In [9]: sigma = np.sqrt(np.log(moms.cv**2 + 1))

In [10]: fzl = ss.lognorm(sigma, scale=moms.ex*np.exp(-sigma**2/2))

In [11]: fzn.sf(1.4 * moms.ex), fzl.sf(1.4 * moms.ex)
Out[11]: (0.1340631332467369, 0.1279965072394511)
```

Notes.

1. How to make the lognormal…

Group Dental Insurance, Examples 9.5, 9.6

Under a group dental insurance plan covering employees and their families, the premium for each married employee is the same regardless of the number of family members. The insurance company has compiled statistics showing that the annual cost of dental care per person for the benefits provided by the plan has the distribution (given in units of 25) on the left.

Furthermore, the distribution of the number of persons per insurance certificate (i.e. per employee) receiving dental care in any year has the distribution on the right.

Determine the mean and standard deviation of total payments per employee.

x	$f(x)$	n	$\Pr(N=n)$
1	0.150	0	0.05
2	0.200	1	0.10
3	0.250	2	0.15
4	0.125	3	0.20
5	0.075	4	0.25
6	0.050	5	0.15
7	0.050	6	0.06
8	0.050	7	0.03
9	0.025	8	0.01
10	0.025		

```
In [12]: kpw_9_5 = build('agg KPW.95 '
   ....:                 'dfreq  [0:8] [0.05, 0.1, 0.15, 0.2, 0.25, 0.15, 0.06, 0.
↪03, 0.01] '
   ....:                 'dsev [1:10] [0.15, 0.2, 0.25, 0.125, 0.075, 0.05, 0.05,␣
↪0.05, 0.025, 0.025]')
   ....:

In [13]: qd(kpw_9_5)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   3.4                      0.50602           0.063622
Sev    3.7      3.7           0 0.62572   0.62572   1.0074      1.0074
Agg  12.58    12.58 -4.4409e-16 0.60927   0.60927  0.52196     0.52196
log2 = 8, bandwidth = 1, validation: not unreasonable.

In [14]: mv(kpw_9_5)
mean     = 12.58
variance = 58.7464
std dev  = 7.66462
```

The probability distributions are in the `density_df` dataframe.

```
In [15]: with pd.option_context('display.max_rows', 360, 'display.max_columns', 10,
   ....:                        'display.width', 150,
   ....:                        'display.float_format', lambda x: f'{x:.5g}'):
   ....:     print(kpw_9_5.density_df.query('p > 0')[['p', 'F', 'S']])
   ....:
              p        F          S
loss
0          0.05     0.05       0.95
1         0.015    0.065      0.935
2      0.023375 0.088375    0.91163
3      0.034675  0.12305    0.87695
4      0.032577  0.15563    0.84437
5      0.035786  0.19141    0.80859
6      0.039808  0.23122    0.76878
7      0.043562  0.27478    0.72522
```

(continues on next page)

(continued from previous page)

```
8      0.047518   0.3223     0.6777
9      0.049034  0.37133    0.62867
10     0.051898  0.42323    0.57677
11     0.051379  0.47461    0.52539
12     0.051187   0.5258     0.4742
13     0.050305   0.5761     0.4239
14     0.048182  0.62429    0.37571
15     0.045759  0.67004    0.32996
16     0.042809  0.71285    0.28715
17     0.039378  0.75223    0.24777
18     0.035746  0.78798    0.21202
19     0.031968  0.81995    0.18005
20     0.028324  0.84827    0.15173
21     0.024788  0.87306    0.12694
22     0.021491  0.89455    0.10545
23     0.018458  0.91301   0.086993
24     0.015692   0.9287     0.0713
25     0.013234  0.94193   0.058066
26     0.011076  0.95301    0.04699
27    0.0092013  0.96221   0.037789
28    0.0075941  0.96981   0.030195
29    0.0062231  0.97603   0.023972
30    0.0050662  0.98109   0.018906
31    0.0040954  0.98519    0.01481
32     0.003288  0.98848   0.011522
33    0.0026221   0.9911  0.0089001
34     0.002076  0.99318  0.0068241
35     0.001632  0.99481  0.0051921
36    0.0012732  0.99608  0.0039189
37   0.00098531  0.99707  0.0029336
38   0.00075628  0.99782  0.0021773
39   0.00057544   0.9984  0.0016019
40     0.000434  0.99883  0.0011679
41   0.00032431  0.99916 0.00084357
42   0.00024008   0.9994 0.00060349
43     0.000176  0.99957 0.00042749
44   0.00012773   0.9997 0.00029976
45   9.1748e-05  0.99979 0.00020801
46     6.52e-05  0.99986 0.00014281
47   4.5831e-05   0.9999 9.6977e-05
48   3.1858e-05  0.99993 6.5119e-05
49   2.1894e-05  0.99996 4.3225e-05
50   1.4871e-05  0.99997 2.8353e-05
51   9.9806e-06  0.99998 1.8373e-05
52   6.6161e-06  0.99999 1.1757e-05
53   4.3305e-06  0.99999 7.4261e-06
54   2.7976e-06        1 4.6285e-06
55   1.7833e-06        1 2.8452e-06
56   1.1211e-06        1 1.7241e-06
57   6.9482e-07        1 1.0292e-06
58   4.2428e-07        1 6.0496e-07
59   2.5511e-07        1 3.4985e-07
60   1.5095e-07        1  1.989e-07
61   8.7827e-08        1 1.1107e-07
62   5.0209e-08        1  6.086e-08
63   2.8179e-08        1 3.2681e-08
64   1.5508e-08        1 1.7173e-08
65   8.3573e-09        1 8.8161e-09
66   4.4032e-09        1 4.4129e-09
67   2.2637e-09        1 2.1492e-09
68   1.1334e-09        1 1.0158e-09
```

(continues on next page)

(continued from previous page)

```
69   5.5143e-10        1 4.6435e-10
70   2.6002e-10        1 2.0433e-10
71   1.1836e-10        1 8.5974e-11
72   5.1711e-11        1 3.4264e-11
73   2.1497e-11        1 1.2767e-11
74   8.3923e-12        1 4.3749e-12
75   3.0273e-12        1 1.3476e-12
76   9.8572e-13        1 3.6182e-13
77   2.8076e-13        1 8.1046e-14
78   6.7141e-14        1 1.3878e-14
79    1.221e-14        1 1.6653e-15
80   1.5255e-15        1 1.1102e-16
```

Aggregate stop loss premiums can be computed as tail integrals of the survival function. Multiply by the units, 25.

```
In [16]: (kpw_9_5.density_df.S[::-1].cumsum()[::-1] * 25)[:8]
Out[16]:
loss
0.0    314.500000
1.0    290.750000
2.0    267.375000
3.0    244.584375
4.0    222.660625
5.0    201.551289
6.0    181.336613
7.0    162.117133
8.0    143.986712
Name: S, dtype: float64
```

Exercise 9.19. An insurance portfolio produces N = 0, 1, 3 claims with probabilities 0.5, 0.4, 0.1. Individual claim amounts are 1 or 10 with probability 0.9, 0.1. Individual claim amounts and N are mutually independent. Calculate the probability that the ratio of aggregate claims to expected claims will exceed 3.0.

```
In [17]: kpw_9_19 = build('agg KPW.9.19 dfreq [0 1 3] [.5 .4 .1] '
   ....:                  'dsev [1 10] [.9 .1]')
   ....:

In [18]: qd(kpw_9_19)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq   0.7                      1.2857              1.4486
Sev    1.9      1.9           0 1.4211    1.4211   2.6667      2.6667
Agg   1.33     1.33 -9.992e-16 2.1302    2.1302    3.414       3.414
log2 = 6, bandwidth = 1, validation: not unreasonable.

In [19]: m = kpw_9_19.agg_m

In [20]: print(f'mean        {m:.5g}\nprobability {kpw_9_19.sf(3 * m):.4g}')
mean        1.33
probability 0.0671
```

Exercise 9.23. An individual loss distribution is normal with mean = 100 and variance = 9. The distribution for the number of claims has outcomes 0, 1, 2, 3 with probabilities 0.5, 0.2, 0.2, 0.1. Determine the probability that aggregate claims exceed 100.

```
In [21]: kpw_9_23 = build('agg KPW.9.23 dfreq [0:3] [1/2 1/5 1/5 1/10] '
   ....:                  'sev 3 * norm + 100')
   ....:

In [22]: qd(kpw_9_23)
```

(continues on next page)

(continued from previous page)

```
      E[X] Est E[X]    Err E[X]  CV(X) Est CV(X)    Skew(X) Est Skew(X)
X
Freq   0.9                        1.16               0.7276
Sev    100       100 -1.1102e-16   0.03      0.03 -5.174e-11           0
Agg     90        90 -1.1102e-15 1.1605    1.1605    0.72937     0.72937
log2 = 16, bandwidth = 1/32, validation: fails sev skew.

In [23]: qd(kpw_9_23.density_df.loc[90:110:64, ['p', 'F', 'S']])

               p       F       S
loss
90.0  3.2132e-06 0.50009 0.49991
92.0  2.3742e-05 0.50078 0.49922
94.0  0.00011248 0.50461 0.49539
96.0  0.00034169 0.51841 0.48159
98.0  0.00066552 0.55083 0.44917
100.0 0.00083113 0.60042 0.39958
102.0 0.00066552 0.64983 0.35017
104.0 0.00034169 0.68193 0.31807
106.0 0.00011248 0.69551 0.30449
108.0 2.3742e-05 0.69925 0.30075
110.0 3.2132e-06 0.69992 0.30008
```

Exercise 9.24. An employer self-insures a life insurance program with the following two characteristics:

1. Given that a claim has occurred, the claim amount is 2,000 with probability 0.4 and 3,000 with probability 0.6.
2. The number of claims has outcomes 0, 1, 2, 3, 4 with probabilities 1/16, 1/4, 3/8, 1/4, 1/16.

The employer purchases aggregate stop-loss coverage that limits the employer's annual claims cost to 5,000. The aggregate stop-loss coverage costs 1,472. Determine the employer's expected annual cost of the program, including the cost of stop-loss coverage.

```
In [24]: kpw_9_24 = build('agg KPW.9.24 dfreq [0:4] [1/16 1/4 3/8 1/4 1/16] '
   ....:                  'dsev [2 3] [0.4 0.6] '
   ....:                  'aggregate net of inf xs 5')
   ....:

In [25]: qd(kpw_9_24)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq     2                       0.5                  0
Sev    2.6      2.6        0 0.18842   0.18842 -0.40825    -0.40825
Agg    5.2   4.0275 -0.22548 0.51745   0.36633 0.091166     -1.4019
log2 = 5, bandwidth = 1, validation: n/a, reinsurance.

In [26]: net = kpw_9_24.describe.iloc[-1, 1]

In [27]: print(f'\ngross loss     {kpw_9_24.agg_m:.5g}\nretained loss {net:.5g}\n'
   ....:       f'premium       {net + 1.472:.5g}')
   ....:

gross loss    5.2
retained loss 4.0275
premium       5.4995
```

Working in thousands.

Exercise 9.31. Medical and dental claims are assumed to be independent with compound Poisson distributions as follows:

- Medical claims 2 expected claims, amounts uniform (0, 1000)
- Dental claims 3 expected claims, amounts uniform (0, 200)

Let X be the amount of a given claim under a policy that covers both medical and dental claims. Determine E[(X − 100)+], the expected cost (in excess of 100) of any given claim.

```
In [28]: kpw_9_31 = build('agg KPW.9.31 [2 3] claims '
   ....:                  'sev [1000 200] * uniform '
   ....:                  'occurrence ceded to inf xs 100 '
   ....:                  'poisson')
   ....:

In [29]: qd(kpw_9_31)

      E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     5                   0.44721            0.44721
Sev    260      177 -0.31923  1.0444     1.461   1.3042      1.4244
Agg   1300      885 -0.31923 0.64664   0.79177  0.85178     0.95458
log2 = 16, bandwidth = 1/4, validation: n/a, reinsurance.

In [30]: qd(kpw_9_31.reinsurance_audit_df.stack(0).head(3))

                                 ex    var     sd      cv    skew
kind share limit attach
occ  1.0   inf   100.0  ceded   177  66871 258.59   1.461  1.4244
                        net      83 844.34 29.057 0.35009 -1.5092
                        subject 260  73733 271.54  1.0444  1.3042
```

Could also compute impact of aggregate reinsurance structures.

Exercise 9.34. A compound Poisson distribution has 5 expected claim and claim amount distribution p(100) = 0.80, p(500) = 0.16, and p(1,000) = 0.04. Determine the probability that aggregate claims will be exactly 600.

```
In [31]: kpw_9_34 = build('agg KPW.9.34 5 claims '
   ....:                  'dsev [1 5 10] [.8 .16 .04] '
   ....:                  'poisson')
   ....:

In [32]: qd(kpw_9_34)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     5                      0.44721            0.44721
Sev      2        2 -2.2204e-16  1.0954    1.0954   2.2822      2.2822
Agg     10       10  4.6629e-15 0.66332   0.66332   1.0416      1.0416
log2 = 10, bandwidth = 1, validation: not unreasonable.

In [33]: print(f'{kpw_9_34.pmf(6):.6g}')
0.0598929

In [34]: kpw_9_34.density_df.index = kpw_9_34.density_df.index.astype(int)

In [35]: qd(kpw_9_34.density_df.query('p > 0.001')[['p', 'F', 'S']], accuracy=5)

             p         F         S
loss
0    0.0067379 0.0067379   0.99326
1     0.026952   0.03369   0.96631
2     0.053904  0.087593   0.91241
3     0.071871   0.15946   0.84054
4     0.071871   0.23134   0.76866
```

(continues on next page)

(continued from previous page)

```
5     0.062888   0.29422   0.70578
6     0.059893   0.35412   0.64588
7     0.065027   0.41914   0.58086
8     0.068449   0.48759   0.51241
9     0.062365   0.54996   0.45004
10    0.051448   0.60141   0.39859
11    0.045388   0.64679   0.35321
...        ...       ...       ...
23   0.0099885   0.95791  0.042093
24   0.0084688   0.96638  0.033624
25   0.0065328   0.97291  0.027091
26   0.0050132   0.97792  0.022078
27   0.0042014   0.98212  0.017877
28   0.0036975   0.98582  0.014179
29   0.0030698   0.98889  0.011109
30   0.0023339   0.99122 0.0087753
31   0.0017543   0.99298  0.007021
32   0.0014285   0.99441 0.0055925
33   0.0012267   0.99563 0.0043658
34   0.0010036   0.99664 0.0033621
```

Work in hundreds. Convert index to integer to improve display. Show all outcomes with probability greater than 0.001.

Exercise 9.35. Aggregate payments have a compound distribution. The frequency distribution is negative binomial with $r = 16$ and $\beta = 6$, and the severity distribution is uniform on the interval (0, 8). Use the normal approximation to determine the premium such that the probability is 5% that aggregate payments will exceed the premium.

The negative binomial has mean $r\beta$ and variance $r\beta(1 + \beta)$. Therefore the gamma mixing variance equals $c = 1/r$ (since $r\beta(1 + \beta) = n(1 + cn)$.) Hence the mixing cv equals 0.25. The premium is the 95%ile of the aggregate distribution.

```
In [36]: kpw_9_35 = build('agg KPW.9.35 96 claims '
   ....:                  'sev 8 * uniform '
   ....:                  'mixed gamma 0.25')
   ....:

In [37]: qd(kpw_9_35)

      E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq    96                       0.27003             0.50149
Sev      4        4           0 0.57735   0.57736          0           0
Agg    384      384 3.3307e-15 0.27639   0.27639   0.50366     0.50366
log2 = 16, bandwidth = 1/32, validation: not unreasonable.

In [38]: mv(kpw_9_35)
mean     = 384
variance = 11264
std dev  = 106.132

In [39]: appx = kpw_9_35.approximate('all')

In [40]: ans = {k: v.isf(0.05) for k, v in appx.items()}

In [41]: ans['FFT'] = kpw_9_35.q(0.95)

In [42]: qd(pd.DataFrame(ans.values(),
   ....:                 index=pd.Index(ans.keys(), name='method'),
   ....:                 columns=['premium']).sort_values('premium'),
   ....:    accuracy=4)
```

(continues on next page)

(continued from previous page)

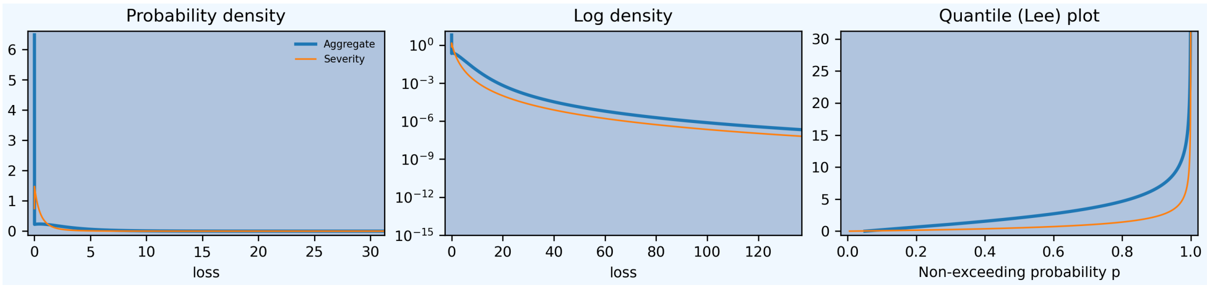
```
   ....:

          premium
method
norm       558.57
slognorm   571.88
sgamma      572.4
FFT        572.41
gamma       573.6
lognorm    578.31
```

The `approximate` method returns a dictionary with key the method, for normal and shifted and unshifted gamma and lognormal.

Exercise 9.36. The number of losses is Poisson with mean 3. Loss amounts have a Burr distribution with $\alpha = 3$, $\theta = 2$, and $\gamma = 1$. Determine the variance of aggregate losses.

A matter of converting parameterizations. This is the `scipy.stats` `burr12` distribution. The shape parameters are `c=gamma` and `d=alpha`. `theta` is a scale parameter.

```
In [43]: kpw_9_36 = build('agg KPW.9.36 3 claims '
   ....:                  'sev 2 * burr12 1 3 '
   ....:                  'poisson')
   ....:

In [44]: qd(kpw_9_36)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     3                      0.57735            0.57735
Sev      1  0.99995 -4.9059e-05  1.7321    1.7187      inf      15.681
Agg      3   2.9999 -4.9063e-05  1.1547     1.148      inf      6.5701
log2 = 16, bandwidth = 1/128, validation: fails sev cv, agg cv.

In [45]: mv(kpw_9_36)
mean     = 3
variance = 12
std dev  = 3.4641

In [46]: kpw_9_36.plot()
```



Compound Poisson, Example 9.9, 9.10

Policy A has a compound Poisson distribution with 2 expected claims and severity probabilities 0.6 on a payment of 1 and 0.4 on a payment of 2. Policy B has a compound Poisson distribution with 1 expected claim and probabilities 0.7 on a payment of 1 and 0.3 on a payment of 3.

Determine the probability that the total payment on the two policies will be 2.

Figure the weighted severity by hand.

```
In [47]: kpw_9_9 = build('agg KPW.9.9 3 claims '
   ....:                 'dsev [1 2 3] [1.9/3 .8/3 .3/3] '
   ....:                 'poisson')
   ....:

In [48]: qd(kpw_9_9)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq      3                      0.57735            0.57735
Sev  1.4667   1.4667 -1.1102e-16 0.45681   0.45681   1.1192      1.1192
Agg     4.4      4.4           0 0.63474   0.63474  0.75284     0.75284
log2 = 7, bandwidth = 1, validation: not unreasonable.

In [49]: print(f'{kpw_9_9.pmf(2):.6g}')
0.129695

In [50]: kpw_9_9.density_df.index = kpw_9_9.density_df.index.astype(int)

In [51]: bit = kpw_9_9.density_df.query('p > 0.001')[['p', 'F', 'S']]

In [52]: bit['p*exp(3)'] = bit.p * np.exp(3)

In [53]: qd(bit, accuracy=5)

             p        F         S  p*exp(3)
loss
0     0.049787 0.049787   0.95021         1
1     0.094595  0.14438   0.85562       1.9
2       0.1297  0.27408   0.72592     2.605
3      0.14753  0.42161   0.57839    2.9632
4      0.14324  0.56484   0.43516     2.877
5      0.12498  0.68983   0.31017    2.5104
6     0.099904  0.78973   0.21027    2.0066
7     0.074101  0.86383   0.13617    1.4884
8     0.051641  0.91547  0.084527    1.0372
9     0.034066  0.94954  0.050462   0.68423
10    0.021404  0.97094  0.029058   0.42991
11    0.012877  0.98382   0.01618   0.25865
12   0.0074477  0.99127 0.0087326   0.14959
13   0.0041552  0.99542 0.0045774   0.08346
14   0.0022429  0.99767 0.0023345   0.04505
15   0.0011742  0.99884 0.0011603  0.023584
```

The last column answers Example 9.10.

Alternatively, use the `Portfolio` class.

```
In [54]: p = build('port KPW.9.9.p '
   ....:           'agg A 2 claims dsev [1 2] [.6 .4] poisson '
   ....:            'agg B 1 claims dsev [1 3] [.7 .3] poisson')
   ....:
```

(continues on next page)

(continued from previous page)

```
In [55]: qd(p)

             E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit  X
A     Freq      2                      0.70711            0.70711
      Sev     1.4      1.4           0 0.34993   0.34993  0.40825     0.40825
      Agg     2.8      2.8  6.6613e-16 0.74915   0.74915  0.82344     0.82344
B     Freq      1                            1                  1
      Sev     1.6      1.6  2.2204e-16 0.57282   0.57282  0.87287     0.87287
      Agg     1.6      1.6  8.8818e-16  1.1524    1.1524   1.4037      1.4037
total Freq      3                      0.57735            0.57735
      Sev  1.4667   1.4667  2.2204e-16 0.45681             1.1192
      Agg     4.4      4.4 -1.6653e-15 0.63474   0.63474  0.75284     0.75284
log2 = 16, bandwidth = 1/1024, validation: not unreasonable.
```

Exercise 9.39. For a compound distribution, frequency has a binomial distribution with parameters m = 3 and q = 0.4 and severity has an exponential distribution with a mean of 100. Calculate $\Pr(A \le 300)$.

Assume 1.2 expected claims. Work in hundreds.

```
In [56]: kpw_9_39 = build('agg KPW.9.39 1.2 claims '
   ....:                  'sev expon binomial 0.4')
   ....:

In [57]: qd(kpw_9_39)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   1.2                      0.70711             0.2357
Sev      1        1 -3.9736e-08       1         1        2           2
Agg    1.2      1.2 -3.9736e-08  1.1547    1.1547   1.7681      1.7681
log2 = 16, bandwidth = 1/1024, validation: not unreasonable.

In [58]: print(f'probability = {kpw_9_39.cdf(3):.6g}')
probability = 0.894092
```

Exercise 9.40. A company sells three policies. For policy A, all claim payments are 10,000 and a single policy has a Poisson number of claims with mean 0.01. For policy B, all claim payments are 20,000 and a single policy has a Poisson number of claims with mean 0.02. For policy C, all claim payments are 40,000 and a single policy has a Poisson number of claims with mean 0.03. All policies are independent. For the coming year, there are 5,000, 3,000, and 1,000 of policies A, B, and C, respectively. Calculate the expected total payment, the standard deviation of total payment, and the probability that total payments will exceed 30,000.

Must use a `Portfolio`. Work in thousands.

```
In [59]: kpw_9_40 = build('port kpw_9_40\n'
   ....:                  '\tagg A 50 claims dsev [10] poisson\n'
   ....:                  '\tagg B 60 claims dsev [20] poisson\n'
   ....:                  '\tagg C 30 claims dsev [40] poisson\n')
   ....:

In [60]: qd(kpw_9_40)

             E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit  X
A     Freq     50                       0.14142            0.14142
      Sev      10       10           0        0         0
      Agg     500      500  1.9984e-15  0.14142   0.14142  0.14142     0.14142
B     Freq     60                        0.1291             0.1291
      Sev      20       20           0        0         0
      Agg    1200     1200 -6.1062e-15   0.1291    0.1291   0.1291      0.1291
```

(continues on next page)

(continued from previous page)

```
C     Freq     30                      0.18257           0.18257
      Sev      40       40           0       0          0
      Agg    1200     1200 -2.2204e-15  0.18257   0.18257  0.18257      0.18257
total Freq    140                     0.084515          0.084515
      Sev  20.714   20.714           0  0.53086            0.82553
      Agg    2900   2899.9 -4.1178e-05 0.095686  0.095811  0.11466      0.08019
log2 = 16, bandwidth = 1/16, validation: fails agg cv.

In [61]: qd(pd.Series({'expected payment': kpw_9_40.agg_m,
   ....:               'sd payment': kpw_9_40.agg_sd,
   ....:               'Pr > 3000': kpw_9_40.sf(3000)}).to_frame('value'),
   ....:               accuracy=5)
   ....:

                   value
expected payment    2900
sd payment        277.49
Pr > 3000        0.34658
```

ZM Binomial, Example 9.11

A compound distribution has a zero-modified binomial distribution with *m* = 3, $q = 0.3$, and $p_0^M = 0.4$. Individual payments are 0, 50, and 150, with probabilities 0.3, 0.5, and 0.2, respectively. Use the recursive formula to determine the probability distribution of S.

Todo: Implement ZM and ZT.

ETNB, Example 9.12

The number of claims has a Poisson–ETNB distribution with Poisson parameter *λ* = 2 and ETNB parameters $\beta = 3$ and $r = 0.2$. The claim size distribution has probabilities 0.3, 0.5, and 0.2 at 0, 10, and 20, respectively. Determine the total claims distribution recursively.

Todo: Implement ZM and ZT.

Exercise 9.45. For a compound Poisson distribution, has 6 expected claims and individual losses take values 1, 2, 4 with equal probabilities. Determine the distribution of the aggregate.

```
In [62]: kpw_9_45 = build('agg KPW.9.45 6 claims '
   ....:                  'dsev [1 2 4] poisson')
   ....:

In [63]: qd(kpw_9_45)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq      6                      0.40825            0.40825
Sev  2.3333   2.3333 2.2204e-16 0.53452   0.53452   0.3818      0.3818
Agg      14       14 4.6629e-15 0.46291   0.46291  0.53639     0.53639
log2 = 9, bandwidth = 1, validation: not unreasonable.

In [64]: qd(kpw_9_45.density_df.query('p > 0.001')[['p', 'F', 'S']], accuracy=5)

             p         F         S
```

(continues on next page)

(continued from previous page)

```
loss
0.0   0.0024788 0.0024788   0.99752
1.0   0.0049575 0.0074363   0.99256
2.0    0.009915  0.017351   0.98265
3.0     0.01322  0.030571   0.96943
4.0    0.021483  0.052054   0.94795
5.0    0.027101  0.079155   0.92085
6.0    0.036575   0.11573   0.88427
7.0    0.041045   0.15678   0.84322
8.0    0.050031   0.20681   0.79319
9.0     0.05345   0.26026   0.73974
10.0   0.059963   0.32022   0.67978
11.0    0.06019   0.38041   0.61959
...         ...       ...       ...
24.0   0.017373   0.93502  0.064978
25.0   0.014016   0.94904  0.050962
26.0   0.011474   0.96051  0.039488
27.0 0.0090615    0.96957  0.030427
28.0 0.0072502    0.97682  0.023176
29.0 0.0056163    0.98244   0.01756
30.0 0.0044008    0.98684  0.013159
31.0 0.0033471    0.99019 0.0098123
32.0 0.0025718    0.99276 0.0072405
33.0 0.0019231    0.99468 0.0053174
34.0 0.0014512    0.99613 0.0038662
35.0 0.0010677     0.9972 0.0027985
```

Exercise 9.47. Aggregate claims are compound Poisson with 2 expected claims and severity outcomes 1, 2 with probability 1/4 and 3/4. For a premium of 6, an insurer covers aggregate claims and agrees to pay a dividend (a refund of premium) equal to the excess, if any, of 75% of the premium over 100% of the claims. Determine the excess of premium over expected claims and dividends.

```
In [65]: kpw_9_47 = build('agg KPW.9.47 2 claims '
   ....:                  'dsev [1 2] [1/4 3/4] poisson')
   ....:

In [66]: qd(kpw_9_47)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     2                      0.70711            0.70711
Sev   1.75     1.75           0 0.24744   0.24744  -1.1547     -1.1547
Agg    3.5      3.5 2.2204e-16 0.72843   0.72843  0.75429     0.75429
log2 = 6, bandwidth = 1, validation: not unreasonable.

In [67]: bit = kpw_9_47.density_df.query('p > 0')[['p', 'F', 'S']]

In [68]: bit['dividend'] = np.maximum(0.75 * 6 - bit.index, 0)

In [69]: qd(bit.head(10), accuracy=4)

            p       F        S  dividend
loss
0.0   0.13534 0.13534  0.86466       4.5
1.0  0.067668   0.203    0.797       3.5
2.0   0.21992 0.42292  0.57708       2.5
3.0   0.10432 0.52724  0.47276       1.5
4.0   0.17798 0.70522  0.29478       0.5
5.0  0.080391 0.78561  0.21439         0
6.0  0.095689  0.8813   0.1187         0
7.0  0.041288 0.92259 0.077408         0
```

(continues on next page)

(continued from previous page)

```
8.0  0.038464 0.96106 0.038945         0
9.0    0.0159 0.97696 0.023045         0

In [70]: exp_div = (bit.dividend * bit.p).sum()

In [71]: print(f'prem      = {6:.5g}\n'
   ....:       f'exp loss  = {kpw_9_47.agg_m:.5g}\n'
   ....:       f'dividend  = {exp_div:.5g}\n'
   ....:       f'excess    = {6 - kpw_9_47.agg_m - exp_div:.5g}')
   ....:
prem      = 6
exp loss  = 3.5
dividend  = 1.6411
excess    = 0.85888
```

Exercise 9.57, 9.58. Aggregate losses have a compound Poisson claim distribution with 3 expected claims and individual claim amount distribution p(1) = 0.4, p(2) = 0.3, p(3) = 0.2, and p(4) = 0.1. Determine the probability that aggregate losses do not exceed 3.

Repeat the Exercise with a negative binomial frequency distribution with r = 6 and $\beta = 0.5$.

```
In [72]: kpw_9_57 = build('agg KPW.9.57 3 claims '
   ....:                  'dsev [1:4] [.4 .3 .2 .1] poisson')
   ....:

In [73]: qd(kpw_9_57)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     3                      0.57735            0.57735
Sev      2        2 -3.3307e-16     0.5       0.5      0.6         0.6
Agg      6        6 -1.1102e-16  0.6455    0.6455  0.75394     0.75394
log2 = 8, bandwidth = 1, validation: not unreasonable.

In [74]: kpw_9_58 = build('agg KPW.9.58 3 claims '
   ....:                  'dsev [1:4] [.4 .3 .2 .1] mixed gamma 6**-0.5')
   ....:

In [75]: qd(kpw_9_58)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     3                      0.70711            0.94281
Sev      2        2 -3.3307e-16     0.5       0.5      0.6         0.6
Agg      6        6 -7.6605e-15 0.76376   0.76376   1.0474      1.0474
log2 = 8, bandwidth = 1, validation: not unreasonable.

In [76]: bit = pd.concat((kpw_9_57.density_df[['p', 'F', 'S']],
   ....:                  kpw_9_58.density_df[['p', 'F', 'S']]),
   ....:                 keys=('Po', 'NB'), axis=1)
   ....:

In [77]: qd(bit.head(16), accuracy=5)

            Po                           NB
             p        F        S        p        F        S
loss
0.0   0.049787 0.049787  0.95021 0.087791 0.087791  0.91221
1.0   0.059744  0.10953  0.89047 0.070233  0.15802  0.84198
2.0   0.080655  0.19019  0.80981  0.08545  0.24348  0.75652
3.0   0.097981  0.28817  0.71183 0.095933  0.33941  0.66059
```

(continues on next page)

(continued from previous page)

```
4.0    0.10751  0.39568  0.60432 0.098486  0.43789  0.56211
5.0    0.10445  0.50013  0.49987 0.089535  0.52743  0.47257
6.0   0.098668   0.5988   0.4012 0.082877  0.61031  0.38969
7.0   0.088215  0.68701  0.31299 0.073657  0.68396  0.31604
8.0    0.07506  0.76207  0.23793 0.063257  0.74722  0.25278
9.0   0.061311  0.82338  0.17662  0.05302  0.80024  0.19976
10.0  0.048587  0.87197  0.12803 0.043819  0.84406  0.15594
11.0  0.037239  0.90921  0.09079 0.035507  0.87957  0.12043
12.0  0.027715  0.93692 0.063075 0.028316  0.90789 0.092115
13.0  0.020101  0.95703 0.042974 0.022288  0.93017 0.069827
14.0  0.014239  0.97127 0.028735 0.017338  0.94751 0.052489
15.0 0.0098562  0.98112 0.018879 0.013334  0.96085 0.039155
```

Exercise 9.59. A policy covers physical damage incurred by the trucks in a company's fleet. The number of losses in a year has a Poisson distribution with expectation 5. The amount of a single loss has a gamma distribution with shape 0.5 and scale 2,500. The insurance contract pays a maximum annual benefit of 20,000. Determine the probability that the maximum benefit will be paid. Use a span of 100 and the method of rounding.

```
In [78]: kpw_9_59 = build('agg KPW.9.59 5 claims '
   ....:                  'sev 2500 * gamma 0.5 '
   ....:                  'poisson')
   ....:

In [79]: qd(kpw_9_59)

      E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq     5                        0.44721             0.44721
Sev   1250     1250 -3.109e-06   1.4142    1.4142    2.8284      2.8284
Agg   6250     6250 -3.109e-06   0.7746    0.7746     1.291       1.291
log2 = 16, bandwidth = 2, validation: not unreasonable.

In [80]: print(f'pr(loss >= 20000) = {kpw_9_59.sf(20000):.6g}')
pr(loss >= 20000) = 0.015939
```

Repeated at the requested span of 100.

```
In [81]: kpw_9_59.update(log2=10, bs=100)

In [82]: print(f'pr(loss >= 20000) = {kpw_9_59.sf(20000):.6g}')
pr(loss >= 20000) = 0.0157042
```

Exercise 9.60. An individual has purchased health insurance, for which they pay 10 for each physician visit and 5 for each prescription. The probability that a payment will be 10 is 0.25, and the probability that it will be 5 is 0.75. The total number of payments per year has the Poisson–Poisson (Neyman Type A) distribution with primary mean 10 and secondary mean 4. Determine the probability that total payments in one year will exceed 400. Compare your answer to a normal approximation.

```
In [83]: kpw_9_60 = build('agg KPW.9.60 40 claims '
   ....:                  'dsev [5 10] [3/4 1/4] '
   ....:                  'neyman 4')
   ....:

In [84]: qd(kpw_9_60)

      E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq    40                        0.35355             0.41012
Sev   6.25     6.25            0 0.34641   0.34641    1.1547      1.1547
Agg    250      250 -9.1038e-14 0.35777   0.35777   0.42101     0.42101
```

(continues on next page)

(continued from previous page)

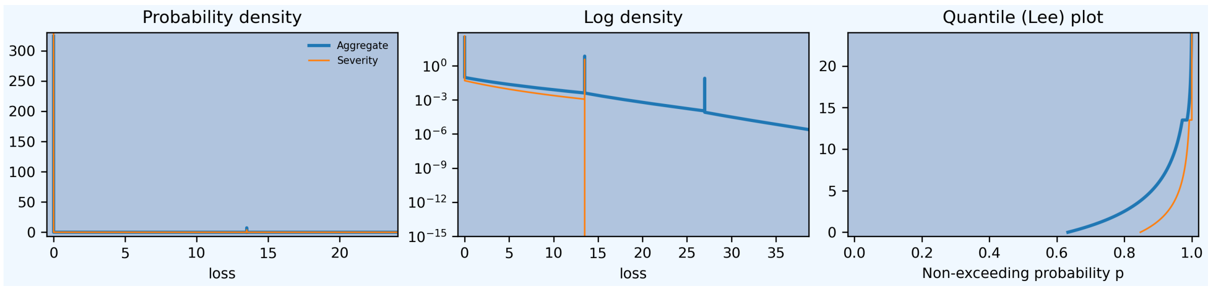
```
log2 = 14, bandwidth = 1, validation: not unreasonable.

In [85]: fz = kpw_9_60.approximate('norm')

In [86]: print(f'FFT            {kpw_9_60.sf(400):.5g}\n'
   ....:       f'Normal approx  {fz.sf(400):.5g}')
   ....:
FFT            0.054616
Normal approx  0.046766
```

Poisson Pareto, Example 9.14

The number of ground-up losses is Poisson distributed with mean 3. The individual loss distribution is Pareto with shape parameter :math:alpha= 4` and scale parameter 10. An individual ordinary deductible of 6, coinsurance of 75%, and an individual loss limit of 24 (before application of the deductible and coinsurance) are all applied. Determine the mean, variance, and distribution of aggregate payments.

The covered layer is 18 xs 6, in which the insured pays 25% because of the coinsurance clause. The severity is unconditional.

```
In [87]: kpw_9_14 = build('agg KPW.9.14 3 claims '
   ....:                  '18 xs 6 '
   ....:                  'sev 10 * pareto 4 - 10 ! '
   ....:                  'occurrence net of 0.25 so inf xs 0 '
   ....:                  'poisson')
   ....:

In [88]: qd(kpw_9_14)

        E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq       3                   0.57735            0.57735
Sev  0.72899  0.54678 -0.24995  3.5115    3.5113   4.6513      4.6511
Agg    2.187   1.6403 -0.24995   2.108    2.1079   2.8396      2.8395
log2 = 16, bandwidth = 1/512, validation: n/a, reinsurance.

In [89]: print(f'variance = {kpw_9_14.describe.iloc[-1,[1, 4]].prod()**2:.6g}\
↪ncomputed with bs=1/{1/kpw_9_14.bs:.0f} and log2={kpw_9_14.log2}')
variance = 11.9552
computed with bs=1/512 and log2=16

In [90]: qd(kpw_9_14.density_df.loc[[0, 1, 2, 3], ['p', 'F', 'S']])

              p       F       S
loss
0.0     0.63277 0.63277 0.36723
1.0  0.00013621 0.71496 0.28504
2.0  0.00010097  0.7751  0.2249
3.0  7.6416e-05 0.82013 0.17987

In [91]: kpw_9_14.plot()
```



`describe` returns gross under `E[X]` and the requested net or ceded under `Est E[X]`. The print statement computes net variance from the product of estimated mean and cv. The spikes on the density corresponds to the possibility of only limit claims. **Exercise 9.63.** A ground-up model of individual losses has a gamma distribution with shape parameter 2 and scale 100. The number of losses has a negative binomial distribution with $r = 2$ and $\beta = 1.5$. An ordinary deductible of 50 and a loss limit of 175 (before imposition of the deductible) are applied to each individual loss.

- Determine the mean and variance of the aggregate payments on a per-loss basis.
- Determine the distribution of the number of payments.
- Determine the cumulative distribution function of the amount of a payment, given that a payment is made.
- Discretize the severity distribution using the method of rounding and a span of 40.
- Calculate the discretized distribution of aggregate payments up to a discretized amount paid of 120.

Negative binomial $c = 1/2$ and hence mixing cv $\sqrt{c}$, and the mean equals $r\beta/(1+\beta) = 1.4$. The cover is 125 xs 50. The severity is unconditional. First, the default calculation using `bs=1/64`.

```
In [92]: kpw_9_63 = build('agg KPW.9.63 1.4 claims '
   ....:                  '125 xs 50 '
   ....:                  'sev 100 * gamma 2 ! '
   ....:                  'mixed gamma 2**-0.5')
   ....:

In [93]: qd(kpw_9_63)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    1.4                       1.1019             1.5557
Sev  86.467   86.467  3.9488e-12 0.54006   0.54006 -0.74977    -0.74977
Agg  121.05   121.05 -3.9543e-08  1.1927    1.1927   1.6385      1.6385
log2 = 16, bandwidth = 1/32, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.

In [94]: mv(kpw_9_63)
mean      = 121.054
variance = 20847.33
std dev   = 144.386

In [95]: qd(kpw_9_63.density_df.loc[:400:40*64,
   ....:     ['p', 'F', 'S', 'p_sev', 'F_sev', 'S_sev']],
   ....:     accuracy=5)
   ....:

               p       F        S      p_sev    F_sev   S_sev
loss
0.0      0.37325 0.37325  0.62675   0.090251 0.090251 0.90975
80.0  4.2253e-05 0.47185  0.52815 0.00011072  0.37323 0.62677
160.0  3.327e-05 0.72045  0.27955          0        1       0
240.0 3.0211e-05 0.80373  0.19627          0        1       0
320.0 1.7591e-05  0.9016 0.098397          0        1       0
400.0 9.0878e-06 0.94991 0.050095          0        1       0
```

Next, calculations performed with the requested broader `bs=40`.

```
In [96]: kpw_9_63.update(log2=8, bs=40)

In [97]: qd(kpw_9_63)

       E[X]  Est E[X]  Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq    1.4                       1.1019             1.5557
Sev  86.467    84.042 -0.02805 0.54006      0.546 -0.74977     -0.81723
Agg  121.05    117.66 -0.02805  1.1927     1.1947   1.6385        1.636
log2 = 8, bandwidth = 40, validation: fails sev mean, agg mean.

In [98]: qd(kpw_9_63.density_df.loc[:400,
   ....:     ['p', 'F', 'S', 'p_sev', 'F_sev', 'S_sev']],
   ....:     accuracy=5)
   ....:

              p       F        S   p_sev   F_sev   S_sev
loss
0.0     0.39509 0.39509  0.60491  0.1558  0.1558  0.8442
40.0    0.05047 0.44556  0.55444 0.14517 0.30097 0.69903
80.0   0.053928 0.49949  0.50051  0.1412 0.44217 0.55783
120.0   0.20376 0.70325  0.29675 0.55783       1       0
160.0  0.042972 0.74622  0.25378       0       1       0
200.0  0.042194 0.78841  0.21159       0       1       0
240.0  0.081719 0.87013  0.12987       0       1       0
280.0  0.024403 0.89453  0.10547       0       1       0
320.0   0.02234 0.91687 0.083127       0       1       0
360.0  0.030181 0.94705 0.052946       0       1       0
400.0  0.011572 0.95863 0.041374       0       1       0
```

The apparent difference in the severity distribution is caused by the rounding method. In the first case F(40) is almost exact whereas in the second it is actually F(60).

Group Life Individual Risk Model, Example 9.15, 9.18

Consider a group life insurance contract with an accidental death benefit. Assume that for all members the probability of death in the next year is 0.01 and that 30% of deaths are accidental. For 50 employees, the benefit for an ordinary death is 50,000 and for an accidental death it is 100,000. For the remaining 25 employees, the benefits are 75,000 and 150,000, respectively. Develop an individual risk model and determine its mean and variance.

The `Portfolio` solution, working in thousands.

```
In [99]: kpw_9_15p = build('port KPW.9.15.p '
   ....:                   'agg A 0.5 claims '
   ....:                       'dsev [50 100] [0.7 0.3] '
   ....:                       'binomial 0.01 '
   ....:                   'agg B 0.25 claims '
   ....:                       'dsev [75 150] [0.7 0.3] '
   ....:                       'binomial 0.01 ')
   ....:

In [100]: qd(kpw_9_15p)

             E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
unit  X
A     Freq    0.5                         1.4071              1.3929
      Sev      65        65           0  0.35251    0.35251   0.87287      0.87287
      Agg    32.5      32.5    2.82e-14   1.4928     1.4928    1.6562       1.6562
B     Freq   0.25                           1.99              1.9699
```

(continues on next page)

(continued from previous page)

```
      Sev    97.5     97.5           0 0.35251   0.35251  0.87287      0.87287
      Agg  24.375   24.375  1.1613e-13  2.1112    2.1112   2.3423       2.3423
total Freq   0.75                       1.1489              1.1373
      Sev  75.833   75.833           0 0.41249              1.1623
      Agg  56.875   56.875 -8.6597e-15  1.2435    1.2435   1.4369       1.4369
log2 = 16, bandwidth = 1/16, validation: not unreasonable.

In [101]: mv(kpw_9_15p)
mean     = 56.875
variance = 5001.984
std dev  = 70.7247
```

The density_df dataframe contains the exact aggregate distribution, which is not easy to compute by other means. KPW says (emphasis added)

> With regard to calculating the probabilities, there are at least three options. One is to do an **exact calculation**, which involves **numerous convolutions** and **almost always requires more excessive computing time**. Recursive formulas have been developed, but they are cumbersome and are not presented here. For one such method, see De Pril [27]. One alternative is a parametric approximation as discussed for the collective risk model. Another alternative is to replace the individual risk model with a similar collective risk model and then do the calculations with that model. These two approaches are presented here.

The following solution attempts to commute convolution through the mixture. This works for a compound Poisson. However, the sum of binomials is not binomial, and so the frequencies can't be independent binomial. They can be independent Poisson because it is additive.

```
In [102]: kpw_9_15w = build('agg KPW.9.15.w '
   .....:                   '0.75 claims '
   .....:                   'dsev [50 75 100 150] '
   .....:                   '[0.35/0.75, 0.175/0.75, 0.15/0.75, 0.075/0.75] '
   .....:                   'binomial 0.01 ')
   .....:

In [103]: qd(kpw_9_15w)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   0.75                       1.1489             1.1373
Sev  75.833   75.833  2.2204e-16 0.41249   0.41249   1.1623      1.1623
Agg  56.875   56.875 -6.9356e-11  1.2437    1.2437   1.4389      1.4389
log2 = 10, bandwidth = 1, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.

In [104]: mv(kpw_9_15w)
mean     = 56.875
variance = 5003.745
std dev  = 70.7372
```

The compound Poisson approximation matches the mean but its variance is slightly off.

```
In [105]: kpw_9_15cp = build('agg KPW.9.15.cp '
   .....:                   '0.75 claims '
   .....:                   'dsev [50 75 100 150] '
   .....:                   '[0.35/0.75, 0.175/0.75, 0.15/0.75, 0.075/0.75] '
   .....:                   'poisson ')
   .....:

In [106]: qd(kpw_9_15cp)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
```

(continues on next page)

(continued from previous page)

```
X
Freq   0.75                       1.1547             1.1547
Sev  75.833   75.833  2.2204e-16 0.41249   0.41249   1.1623      1.1623
Agg  56.875   56.875 -1.1396e-10  1.2491    1.2491   1.4523      1.4523
log2 = 10, bandwidth = 1, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.

In [107]: mv(kpw_9_15cp)
mean      = 56.875
variance = 5046.875
std dev   = 71.0414
```

Comparing probabilities shows that all three distributions are very close.

```
In [108]: bit = pd.concat((kpw_9_15p.density_df.loc[:400:128, ['p_total']].query(
↪'p_total > 1e-10'),
   .....:                    kpw_9_15cp.density_df.loc[:400, ['p_total']].query('p_
↪total > 0'),
   .....:                    kpw_9_15w.density_df.loc[:400, ['p_total']].query('p_
↪total > 0'),
   .....:                   ),
   .....:                   keys=('exact', 'compound Po', 'wrong'), axis=1).
↪rename(columns={'p_total': 'p'})
   .....:

In [109]: bit = bit.droplevel(1, axis=1)

In [110]: bit.index.name = 'loss'

In [111]: qd(bit, accuracy=5)

           exact  compound Po      wrong
loss
0.0      0.47059      0.47237    0.47059
200.0   0.024856     0.024881   0.024823
400.0 0.00061615   0.00064447 0.00061581
50.0         NaN      0.16533    0.16637
75.0         NaN     0.082664   0.083185
100.0        NaN     0.099787    0.10032
125.0        NaN     0.028932   0.029017
150.0        NaN     0.070835   0.071104
175.0        NaN     0.017463   0.017428
225.0        NaN     0.011552   0.011478
250.0        NaN     0.011399     0.0113
275.0        NaN    0.0040587  0.0039712
300.0        NaN    0.0050707   0.004986
325.0        NaN    0.0018166  0.0017595
350.0        NaN     0.001694  0.0016392
375.0        NaN   0.00077117 0.00073805
```

Group Life Individual Risk Model, Example 9.16, 9.17

A small manufacturing business has a group life insurance contract on its 14 permanent employees. The actuary for the insurer has selected a mortality table to represent the mortality of the group. Each employee is insured for the amount of his or her salary rounded up to the next 1,000. The group's data are shown in the next table.

Employee	Age	Sex	Benefit	q
1	20	M	$15,000$	0.00149
2	23	M	$16,000$	0.00142
3	27	M	$20,000$	0.00128
4	30	M	$28,000$	0.00122
5	31	M	$31,000$	0.00123
6	46	M	$18,000$	0.00353
7	47	M	$26,000$	0.00394
8	49	M	$24,000$	0.00484
9	64	M	$60,000$	0.02182
10	17	F	$14,000$	0.00050
11	22	F	$17,000$	0.00050
12	26	F	$19,000$	0.00054
13	37	F	$30,000$	0.00103
14	55	F	$55,000$	0.00479

If the insurer adds a 45% relative loading to the net (pure) premium, what are the chances that it will lose money in a given year? Use the normal and lognormal approximations.

In order to make the answer self-contained, the code below includes the data munging to re-create the table, pasted from a pdf.

```
In [112]: data = '''1
   .....: 20
   .....: M
   .....: 15,000
   .....: 0.00149
   .....: 2
   .....: 23
   .....: M
   .....: 16,000
   .....: 0.00142
   .....: 3
   .....: 27
   .....: M
   .....: 20,000
   .....: 0.00128
   .....: 4
   .....: 30
   .....: M
   .....: 28,000
   .....: 0.00122
   .....: 5
   .....: 31
   .....: M
   .....: 31,000
   .....: 0.00123
   .....: 6
   .....: 46
   .....: M
   .....: 18,000
   .....: 0.00353
   .....: 7
   .....: 47
   .....: M
   .....: 26,000
   .....: 0.00394
```

(continues on next page)

(continued from previous page)

```
   .....: 8
   .....: 49
   .....: M
   .....: 24,000
   .....: 0.00484
   .....: 9
   .....: 64
   .....: M
   .....: 60,000
   .....: 0.02182
   .....: 10
   .....: 17
   .....: F
   .....: 14,000
   .....: 0.00050
   .....: 11
   .....: 22
   .....: F
   .....: 17,000
   .....: 0.00050
   .....: 12
   .....: 26
   .....: F
   .....: 19,000
   .....: 0.00054
   .....: 13
   .....: 37
   .....: F
   .....: 30,000
   .....: 0.00103
   .....: 14
   .....: 55
   .....: F
   .....: 55,000
   .....: 0.00479'''
   .....:

In [113]: sdata = data.split('\n')

In [114]: df = pd.DataFrame(zip(*[iter(sdata)]*5),
   .....:                   columns=['Employee', 'Age', 'Sex', 'Benefit', 'q'])
   .....:

In [115]: df.Benefit = df.Benefit.str.replace(',','').astype(float)

In [116]: df.q = df.q.astype(float)

In [117]: df = df.set_index('Employee')

In [118]: qd(df)

         Age Sex  Benefit       q
Employee
1         20   M    15000 0.00149
2         23   M    16000 0.00142
3         27   M    20000 0.00128
4         30   M    28000 0.00122
5         31   M    31000 0.00123
6         46   M    18000 0.00353
7         47   M    26000 0.00394
8         49   M    24000 0.00484
```

(continues on next page)

(continued from previous page)

```
9         64   M    60000 0.02182
10        17   F    14000  0.0005
11        22   F    17000  0.0005
12        26   F    19000 0.00054
13        37   F    30000 0.00103
14        55   F    55000 0.00479

In [119]: print(f'expected claim count = {df.q.sum():.6g}')
expected claim count = 0.04813
```

Here are the FFT-exact, and various approximations to the required probability. Working in thousands. The `dsev` clauses enter the fixed benefit amount for each employee. Note the outsize impact of employee 9.

```
In [120]: from aggregate import Portfolio

In [121]: a = [build(f'agg ee.{i} {r.q} claims '
   .....:            f'dsev [{r.Benefit / 1000}] '
   .....:            f'bernoulli')
   .....:          for i, r in df.iterrows()]
   .....:

In [122]: kpw_9_16p = Portfolio('KPW.9.16p', a)

In [123]: kpw_9_16p.update(log2=8, bs=1, remove_fuzz=True)

In [124]: qd(kpw_9_16p)

              E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit  X
ee.1  Freq 0.00149                        25.887            25.848
      Sev       15       15 -1.1102e-16       0         0
      Agg  0.02235  0.02235  8.8818e-16  25.887    25.887   25.848      25.848
ee.2  Freq 0.00142                        26.518            26.481
      Sev       16       16           0       0         0
      Agg  0.02272  0.02272  -1.481e-13  26.518    26.518   26.481      26.481
ee.3  Freq 0.00128                        27.933            27.897
      Sev       20       20           0       0         0
      Agg   0.0256   0.0256 -8.7597e-14  27.933    27.933   27.897      27.897
ee.4  Freq 0.00122                        28.612            28.577
      Sev       28       28  2.2204e-16       0         0
      Agg  0.03416  0.03416   3.908e-14  28.612    28.612   28.577      28.577
...        ...      ...         ...     ...       ...      ...         ...
ee.12 Freq 0.00054                        43.022            42.998
      Sev       19       19           0       0         0
      Agg  0.01026  0.01026    1.35e-13  43.022    43.022   42.998      42.998
ee.13 Freq 0.00103                        31.143            31.111
      Sev       30       30           0       0         0
      Agg   0.0309   0.0309 -4.2188e-15  31.143    31.143   31.111      31.111
ee.14 Freq 0.00479                        14.414            14.345
      Sev       55       55 -1.1102e-16       0         0
      Agg  0.26345  0.26345 -2.1538e-14  14.414    14.414   14.345      14.345
total Freq 0.04813                        4.5315            4.4789
      Sev   42.685   42.685           0 0.43586          -0.28928
      Agg   2.0544   2.0544 -5.9397e-14  4.9289    4.9289   5.2673      5.2673
log2 = 8, bandwidth = 1, validation: not unreasonable.

In [125]: mv(kpw_9_16p)
mean     = 2.05441
variance = 102.5336
std dev  = 10.1259
```

(continues on next page)

(continued from previous page)

```
In [126]: appx = kpw_9_16p.approximate('all')

In [127]: premium = 1.45 * kpw_9_16p.agg_m

In [128]: ans = {k: v.sf(premium) for k, v in appx.items()}

In [129]: ans['FFT'] = kpw_9_16p.sf(premium)

In [130]: qd(pd.DataFrame(ans.values(),
   .....:                 index=pd.Index(ans.keys(), name='method'),
   .....:                 columns=['premium']).sort_values('premium'),
   .....:    accuracy=5)
   .....:

          premium
method
FFT      0.047261
gamma    0.091498
lognorm   0.13449
sgamma    0.18346
slognorm  0.28099
norm      0.46363
```

Here is a sample from the distribution and the mean-matched compound Poisson (for Exercise 9.18). The latter `dsev` clause works because all the benefit amounts are different. The temporary variable `sev` creates the severity curve. The log pmf graph reflects the irregular benefit amounts. Compare the cdf under `comp Po` with Table 9.17.

```
In [131]: sev = df[['Benefit', 'q']]

In [132]: sev.q = sev.q / sev.q.sum()

In [133]: sev = sev.sort_values('Benefit')

In [134]: kpw_9_16cp = build('agg kpw_9_16.po '
   .....:                    f'{df.q.sum()} claims '
   .....:                    f'dsev {sev.Benefit.values / 1000} {sev.q.values} '
   .....:                    'poisson', bs=1, log2=10)
   .....:

In [135]: qd(kpw_9_16cp)

        E[X] Est E[X]   Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 0.04813                      4.5582             4.5582
Sev   42.685   42.685          0 0.43586   0.43586 -0.28928    -0.28928
Agg   2.0544   2.0544 3.4417e-14  4.9723    4.9723   5.4286      5.4286
log2 = 10, bandwidth = 1, validation: not unreasonable.

In [136]: bit = pd.concat((kpw_9_16p.density_df.query('p_total > 0')[['p_total', 'F
↪', 'S']],
   .....:                  kpw_9_16cp.density_df.query('p_total > 0')[['p_total',
↪'F', 'S']]),
   .....:                 keys=['exact', 'comp Po'], axis=1)
   .....:

In [137]: bit.index = [f'{i:.0f}' for i in bit.index]

In [138]: bit.index.name = 'loss'

In [139]: with pd.option_context('display.max_rows', 360, 'display.max_columns',␣
↪10,
```

(continues on next page)

(continued from previous page)

```
   .....:                        'display.width', 150, 'display.float_format',␣
↪lambda x: f'{x:.7g}'):
   .....:     print(bit)
   .....:
            exact                                 comp Po
          p_total         F            S      p_total         F            S
loss
0        0.952739  0.952739   0.04726095    0.9530099 0.9530099   0.04699011
14   0.0004766078 0.9532157   0.04678434 0.0004765049 0.9534864   0.04651361
15      0.0014217 0.9546374   0.04536264  0.001419985 0.9549064   0.04509362
16    0.001354813 0.9559922   0.04400783  0.001353274 0.9562597   0.04374035
17   0.0004766078 0.9564688   0.04353122 0.0004765049 0.9567362   0.04326384
18    0.003375083 0.9598439   0.04015614  0.003364125 0.9601003   0.03989972
19   0.0005147571 0.9603586   0.03964138 0.0005146252 0.9606149   0.03938509
20    0.001221069 0.9615797   0.03842031  0.001219853 0.9618348   0.03816524
24    0.004633684 0.9662134   0.03378663  0.004612568 0.9664473   0.03355267
26     0.00376864  0.969982   0.03001799  0.003754859 0.9702022   0.02979781
28    0.001163761 0.9711458   0.02885423  0.001162791  0.971365   0.02863502
29   7.112054e-07 0.9711465   0.02885352 7.099922e-07 0.9713657   0.02863431
30   0.0009830108 0.9721295   0.02787051 0.0009833345  0.972349   0.02765098
31    0.001175572 0.9733051   0.02669493  0.001174457 0.9735235   0.02647652
32   2.399591e-06 0.9733075   0.02669253 3.352879e-06 0.9735268   0.02647317
33   5.971631e-06 0.9733134   0.02668656 5.946495e-06 0.9735328   0.02646722
34   6.178405e-06 0.9733196   0.02668038 6.272902e-06 0.9735391   0.02646095
35   4.242488e-06 0.9733239   0.02667614  4.23041e-06 0.9735433   0.02645672
36   1.993891e-06 0.9733259   0.02667415 7.927184e-06 0.9735512   0.02644879
37   2.434369e-06 0.9733283   0.02667171 2.426553e-06 0.9735536   0.02644637
38   6.643644e-06 0.9733349   0.02666507 6.751312e-06 0.9735604   0.02643961
39   7.574225e-06 0.9733425   0.02665749 7.531446e-06 0.9735679   0.02643208
40   8.474451e-06  0.973351   0.02664902 9.207982e-06 0.9735771   0.02642287
41   7.941654e-06 0.9733589   0.02664108 7.901023e-06  0.973585   0.02641497
42    2.23561e-05 0.9733813   0.02661872 2.219562e-05 0.9736072   0.02639278
43   6.125396e-06 0.9733874    0.0266126 6.100774e-06 0.9736133   0.02638668
44   2.143545e-05 0.9734088   0.02659116 2.130123e-05 0.9736346   0.02636538
45   4.672158e-06 0.9734135   0.02658649 4.659237e-06 0.9736393   0.02636072
46   1.210077e-05 0.9734256   0.02657439 1.205366e-05 0.9736513   0.02634866
47   2.791514e-06 0.9734284    0.0265716  2.78804e-06 0.9736541   0.02634587
48    5.56219e-06  0.973434   0.02656603 1.670997e-05 0.9736708   0.02632916
49   4.696495e-06 0.9734387   0.02656134   4.6784e-06 0.9736755   0.02632449
50   2.022647e-05 0.9734589   0.02654111 2.007543e-05 0.9736956   0.02630441
51   1.510358e-06 0.9734604    0.0265396 1.516371e-06 0.9736971   0.02630289
52   5.666162e-06 0.9734661   0.02653394 1.304127e-05 0.9737101   0.02628985
53   1.370555e-08 0.9734661   0.02653392 1.686769e-08 0.9737102   0.02628984
54   9.392317e-06 0.9734755   0.02652453 9.356468e-06 0.9737195   0.02628048
55    0.004591308 0.9780668   0.02193322  0.004570612 0.9782901   0.02170987
56   3.900383e-06 0.9780707   0.02192932  4.60947e-06 0.9782947   0.02170526
57   4.682325e-06 0.9780754   0.02192464 4.659748e-06 0.9782994    0.0217006
58   1.240279e-06 0.9780766    0.0219234 1.246296e-06 0.9783006   0.02169935
59   1.480938e-06 0.9780781   0.02192192 1.478042e-06 0.9783021   0.02169787
60     0.02125253 0.9993306  0.000669383   0.02079525 0.9990974 0.0009026226
61   1.248838e-06 0.9993319 0.0006681341 1.247493e-06 0.9990986 0.0009013751
62   4.172979e-08 0.9993319 0.0006680924  7.93117e-07 0.9990994  0.000900582
63   2.713599e-08 0.9993319 0.0006680653 4.479884e-08 0.9990995 0.0009005372
64   4.244129e-08  0.999332 0.0006680228  6.96877e-08 0.9990995 0.0009004675
65   4.491981e-08  0.999332 0.0006679779 5.030504e-08 0.9990996 0.0009004172
66   4.146533e-08 0.9993321 0.0006679364 9.361181e-08 0.9990997 0.0009003236
67   2.339189e-08 0.9993321  0.000667913 4.760197e-08 0.9990997  0.000900276
68   8.513895e-08 0.9993322 0.0006678279 1.101497e-07 0.9990998 0.0009001659
69   2.329551e-06 0.9993345 0.0006654983 2.321713e-06 0.9991022 0.0008978441
70   6.910878e-06 0.9993414 0.0006585875  6.89662e-06 0.9991091 0.0008909475
71   6.544798e-06  0.999348 0.0006520427 6.512085e-06 0.9991156 0.0008844354
```

(continues on next page)

(continued from previous page)

```
72   2.350356e-06 0.9993503 0.0006496923 2.366974e-06 0.9991179 0.0008820685
73   1.628085e-05 0.9993666 0.0006334115 1.615066e-05 0.9991341 0.0008659178
74   1.314419e-05 0.9993797 0.0006202673 1.294404e-05  0.999147 0.0008529738
75   3.761971e-05 0.9994174 0.0005826475 3.685732e-05 0.9991839 0.0008161165
76   3.023577e-05 0.9994476 0.0005524118 2.959418e-05 0.9992135 0.0007865223
77   1.064505e-05 0.9994582 0.0005417667 1.041214e-05 0.9992239 0.0007761101
78   7.531246e-05 0.9995335 0.0004664543  7.34545e-05 0.9992973 0.0007026556
79   3.379094e-05 0.9995673 0.0004326633 3.334395e-05 0.9993307 0.0006693117
80   2.725768e-05 0.9995946 0.0004054057 2.665243e-05 0.9993573 0.0006426592
81   1.816289e-05 0.9996128 0.0003872428 1.801028e-05 0.9993754  0.000624649
82   5.911614e-09 0.9996128 0.0003872369  1.74019e-08 0.9993754 0.0006246316
83   5.608384e-06 0.9996184 0.0003816285 5.586048e-06  0.999381 0.0006190455
84   0.0001033707 0.9997217 0.0002782578 0.0001006574 0.9994816 0.0005183881
85      4.743e-06 0.9997265 0.0002735148  4.72191e-06 0.9994863 0.0005136662
86   8.972427e-05 0.9998162 0.0001837905 8.756326e-05 0.9995739 0.0004261029
87   1.649612e-08 0.9998162  0.000183774 2.194197e-08 0.9995739  0.000426081
88   2.598867e-05 0.9998422 0.0001577853 2.540443e-05 0.9995993 0.0004006766
89   4.722525e-08 0.9998423 0.0001577381 4.726622e-08 0.9995994 0.0004006293
90   2.194831e-05 0.9998642 0.0001357898 2.147808e-05 0.9996208 0.0003791512
91   2.623287e-05 0.9998904 0.0001095569  2.56655e-05 0.9996465 0.0003534857
92   6.536066e-08 0.9998905 0.0001094916 8.594597e-08 0.9996466 0.0003533998
93    1.65287e-07 0.9998907 0.0001093263  1.62614e-07 0.9996468 0.0003532372
94   1.743665e-07 0.9998908 0.0001091519 1.733535e-07 0.9996469 0.0003530638
95   1.355125e-07  0.999891 0.0001090164 1.365987e-07 0.9996471 0.0003529272
96   8.286137e-08 0.9998911 0.0001089335 2.111788e-07 0.9996473  0.000352716
97   1.619798e-07 0.9998912 0.0001087716 1.594421e-07 0.9996474 0.0003525566
98   1.778121e-07 0.9998914 0.0001085938 1.768561e-07 0.9996476 0.0003523797
99   2.722478e-07 0.9998917 0.0001083215 2.665397e-07 0.9996479 0.0003521132
100  2.116102e-07 0.9998919 0.0001081099 2.234979e-07 0.9996481 0.0003518897
101  2.354722e-07 0.9998921 0.0001078744 2.302657e-07 0.9996483 0.0003516594
102  5.121685e-07 0.9998926 0.0001073623 4.978643e-07 0.9996488 0.0003511616
103   1.63469e-07 0.9998928 0.0001071988 2.132647e-07 0.9996491 0.0003509483
104  5.007765e-07 0.9998933  0.000106698 4.873585e-07 0.9996495 0.0003504609
105  2.016058e-07 0.9998935 0.0001064964 1.979245e-07 0.9996497  0.000350263
106  2.772016e-07 0.9998938 0.0001062192 2.703786e-07   0.99965 0.0003499926
107  8.955276e-08 0.9998939 0.0001061296 1.233815e-07 0.9996501 0.0003498692
108  1.241644e-07  0.999894 0.0001060055   3.6475e-07 0.9996505 0.0003495045
109  1.499989e-07 0.9998941 0.0001058555 1.469601e-07 0.9996506 0.0003493575
110  4.787335e-07 0.9998946 0.0001053768 1.139834e-05  0.999662 0.0003379592
111   5.24875e-08 0.9998947 0.0001053243 5.520826e-08 0.9996621  0.000337904
112  1.489302e-07 0.9998948 0.0001051753 3.069069e-07 0.9996624 0.0003375971
113  6.282775e-09 0.9998948 0.0001051691 6.359186e-09 0.9996624 0.0003375907
114  2.166401e-07  0.999895 0.0001049524 2.112529e-07 0.9996626 0.0003373795
115  0.0001024173 0.9999975   2.53513e-06 9.973353e-05 0.9997624 0.0002376459
116  9.301591e-08 0.9999976 2.442114e-06 1.065637e-07 0.9997625 0.0002375394
117  1.046485e-07 0.9999977 2.337466e-06 1.054805e-07 0.9997626 0.0002374339
118  2.779748e-08 0.9999977 2.309668e-06  2.74138e-08 0.9997626 0.0002374065
119  3.323942e-08 0.9999977 2.276429e-06 3.258744e-08 0.9997626 0.0002373739
120   1.07834e-09 0.9999977  2.27535e-06 0.0002268827 0.9999895 1.049118e-05
121  2.805718e-08 0.9999978 2.247293e-06 2.767002e-08 0.9999895 1.046351e-05
122   1.04354e-09 0.9999978  2.24625e-06 1.753529e-08 0.9999896 1.044597e-05
123  1.015218e-09 0.9999978 2.245234e-06 1.506102e-09 0.9999896 1.044447e-05
124   1.11823e-09 0.9999978 2.244116e-06 7.176094e-09 0.9999896 1.043729e-05
125  1.330164e-09 0.9999978 2.242786e-06  1.78429e-08 0.9999896 1.041945e-05
126  1.040647e-09 0.9999978 2.241745e-06 1.771138e-08 0.9999896 1.040174e-05
127  7.934845e-10 0.9999978 2.240952e-06 6.910456e-09 0.9999896 1.039483e-05
128  2.074415e-09 0.9999978 2.238878e-06 4.117227e-08 0.9999896 1.035365e-05
129  5.213364e-08 0.9999978 2.186744e-06 5.695507e-08 0.9999897  1.02967e-05
130  1.542995e-07  0.999998 2.032444e-06 1.646233e-07 0.9999899 1.013208e-05
131   1.46062e-07 0.9999981 1.886382e-06 1.424089e-07   0.99999 9.989666e-06
132  5.249366e-08 0.9999982 1.833889e-06 5.171857e-08 0.9999901 9.937948e-06
```

(continues on next page)

(continued from previous page)

```
133  3.632949e-07 0.9999985 1.470594e-06 3.526438e-07 0.9999904 9.585304e-06
134   5.60785e-08 0.9999986 1.414515e-06 2.220186e-07 0.9999906 9.363285e-06
135  1.318434e-07 0.9999987 1.282672e-06 4.663595e-07 0.9999911 8.896926e-06
136  4.358886e-10 0.9999987 1.282236e-06 3.667832e-07 0.9999915 8.530142e-06
137  3.292504e-10 0.9999987 1.281907e-06 1.138415e-07 0.9999916 8.416301e-06
138  6.005527e-10 0.9999987 1.281306e-06 8.153441e-07 0.9999924 7.600957e-06
139  4.976509e-07 0.9999992 7.836552e-07 6.051008e-07  0.999993 6.995856e-06
140  4.940901e-10 0.9999992 7.831612e-07 3.024997e-07 0.9999933 6.693356e-06
141  4.051553e-07 0.9999996 3.780058e-07 4.064893e-07 0.9999937 6.286867e-06
142  1.556782e-10 0.9999996 3.778501e-07 4.463796e-10 0.9999937 6.286421e-06
143  1.251058e-07 0.9999997 2.527444e-07 1.219742e-07 0.9999938 6.164447e-06
144  1.937367e-10 0.9999997 2.525506e-07 1.098375e-06 0.9999949 5.066072e-06
145  1.058015e-07 0.9999999 1.467492e-07 1.030876e-07  0.999995 4.962984e-06
146  1.262168e-07         1 2.053237e-08 1.016858e-06 0.9999961 3.946126e-06
147  3.685373e-10         1 2.016383e-08 5.121882e-10 0.9999961 3.945614e-06
148   6.46479e-10         1 1.951735e-08  2.77595e-07 0.9999963 3.668019e-06
149  6.999917e-10         1 1.881736e-08 9.506666e-10 0.9999963 3.667068e-06
150  4.588675e-10         1 1.835849e-08 2.346692e-07 0.9999966 3.432399e-06
151  2.170583e-10         1 1.814144e-08 2.805268e-07 0.9999968 3.151872e-06
152  2.643346e-10         1  1.78771e-08  1.33265e-09 0.9999968  3.15054e-06
153   7.16224e-10         1 1.716088e-08 2.204159e-09 0.9999969 3.148335e-06
154  8.158174e-10         1 1.634506e-08 2.534458e-09 0.9999969 3.145801e-06
155  9.122321e-10         1 1.543283e-08 2.028215e-09 0.9999969 3.143773e-06
156  8.566022e-10         1 1.457623e-08 2.859701e-09 0.9999969 3.140913e-06
157   2.40212e-09         1 1.217411e-08  2.93432e-09 0.9999969 3.137979e-06
158  6.608837e-10         1 1.151322e-08  2.44401e-09 0.9999969 3.135535e-06
159  2.304181e-09         1 9.209041e-09 4.077408e-09 0.9999969 3.131457e-06
160  5.036921e-10         1 8.705349e-09 2.915487e-09 0.9999969 3.128542e-06
161  1.300943e-09         1 7.404406e-09 3.161405e-09 0.9999969  3.12538e-06
162  3.006818e-10         1 7.103724e-09 5.729582e-09 0.9999969 3.119651e-06
163  5.986384e-10         1 6.505086e-09 3.201579e-09 0.9999969 3.116449e-06
164  5.047601e-10         1 6.000326e-09 5.670897e-09 0.9999969 3.110778e-06
165  2.172347e-09         1 3.827979e-09 2.073139e-08 0.9999969 3.090047e-06
166  1.623502e-10         1 3.665628e-09 3.083291e-09 0.9999969 3.086964e-06
167  6.086006e-10         1 3.057028e-09 2.082057e-09 0.9999969 3.084882e-06
168   2.04573e-12         1 3.054982e-09 3.995334e-09 0.9999969 3.080886e-06
169  1.009057e-09         1 2.045925e-09 2.109975e-09 0.9999969 3.078776e-06
170  6.145147e-10         1 1.431411e-09 2.439393e-07 0.9999972 2.834837e-06
171  4.192868e-10         1 1.012124e-09 8.580199e-10 0.9999972 2.833979e-06
172  5.027193e-10         1 5.094045e-10 3.601279e-09 0.9999972 2.830378e-06
173  1.333299e-10         1 3.760746e-10 1.352801e-10 0.9999972 2.830242e-06
174  1.590085e-10         1 2.170661e-10 2.382985e-09 0.9999972 2.827859e-06
175  4.288268e-12         1 2.127779e-10 1.088124e-06 0.9999983 1.739736e-06
176  1.340885e-10         1 7.868939e-11 1.228987e-09 0.9999983 1.738507e-06
177  4.486189e-12         1  7.42032e-11 1.192855e-09 0.9999983 1.737314e-06
178  2.920835e-12         1 7.128231e-11  3.02749e-10 0.9999983 1.737011e-06
179  4.562823e-12         1 6.671952e-11 3.683841e-10 0.9999983 1.736643e-06
180   4.82662e-12         1 6.189294e-11  1.65027e-06 0.9999999 8.637285e-08
181  4.454919e-12         1 5.743805e-11 3.319206e-10 0.9999999 8.604093e-08
182  2.514367e-12         1 5.492373e-11 2.035136e-10 0.9999999 8.583741e-08
183  9.143925e-12         1 4.577982e-11 8.424069e-11 0.9999999 8.575317e-08
184  3.826298e-12         1 4.195355e-11 1.503577e-10 0.9999999 8.560282e-08
185  7.320083e-12         1 3.463352e-11 4.000487e-10 0.9999999 8.520277e-08
186  2.580997e-12         1 3.205247e-11  3.64933e-10 0.9999999 8.483783e-08
187  6.060502e-12         1 2.599199e-11 1.396254e-10 0.9999999 8.469821e-08
188  3.909234e-12         1 2.208278e-11 8.727237e-10 0.9999999 8.382548e-08
189  3.768936e-12         1 1.831379e-11 7.747876e-10 0.9999999  8.30507e-08
190  3.132506e-12         1  1.51813e-11 1.950703e-09 0.9999999 8.109999e-08
191  1.538332e-12         1 1.364298e-11  1.62618e-09 0.9999999 7.947381e-08
192  1.447886e-12         1 1.219513e-11 5.652266e-10 0.9999999 7.890859e-08
193  2.725839e-12         1 9.469314e-12 3.871408e-09 0.9999999 7.503718e-08
```

(continues on next page)

(continued from previous page)

```
194  6.673824e-13         1 8.801959e-12  2.58819e-09 0.9999999 7.244899e-08
195  2.106876e-12         1 6.695089e-12 3.878648e-09 0.9999999 6.857034e-08
196  2.600251e-12         1 4.094836e-12 3.322862e-09 0.9999999 6.524748e-08
197  6.347047e-13         1 3.460121e-12 8.305254e-10 0.9999999 6.441695e-08
198  7.662416e-13         1 2.693845e-12 6.129516e-09 0.9999999 5.828744e-08
199  5.279873e-13         1 2.165823e-12 6.156754e-09 0.9999999 5.213068e-08
200  1.257773e-12         1 9.080514e-13  2.36799e-09         1 4.976269e-08
201  1.318127e-14         1 8.948398e-13 4.582293e-09         1  4.51804e-08
202  5.310998e-13         1 3.637091e-13 5.655193e-12         1 4.517475e-08
203  2.332875e-14         1 3.403944e-13 1.331814e-09         1 4.384293e-08
204  1.743248e-13         1 1.660894e-13 7.991045e-09         1 3.585189e-08
205  1.421973e-14         1 1.518785e-13 1.125462e-09         1 3.472643e-08
206  1.067366e-14         1 1.412204e-13  7.84441e-09         1 2.688202e-08
207   1.25541e-14         1 1.286748e-13 6.361132e-12         1 2.687566e-08
208  1.108523e-14         1 1.175726e-13 2.022885e-09         1 2.485277e-08
209  9.803926e-15         1 1.078027e-13   1.1113e-11         1 2.484166e-08
210  9.503263e-15         1 9.825474e-14 1.710189e-09         1 2.313147e-08
211  1.723964e-14         1 8.104628e-14 2.045013e-09         1 2.108646e-08
212   8.14414e-15         1 7.294165e-14 1.447721e-11         1 2.107198e-08
213  1.419199e-14         1  5.87308e-14  1.99869e-11         1 2.105199e-08
214  1.306909e-14         1 4.563017e-14 2.498278e-11         1 2.102701e-08
215  9.182056e-15         1 3.641532e-14 1.943114e-11         1 2.100758e-08
216    8.3874e-15         1 2.797762e-14 2.588086e-11         1  2.09817e-08
217  4.424326e-15         1 2.353673e-14 3.051195e-11         1 2.095118e-08
218  6.462422e-15         1 1.709743e-14 2.291723e-11         1 2.092827e-08
219  1.852404e-15         1 1.521006e-14  3.87301e-11         1 2.088954e-08
220  3.656445e-15         1 1.154632e-14 4.736282e-11         1 2.084217e-08
221   3.85918e-16         1 1.121325e-14 2.793092e-11         1 2.081424e-08
222  1.257844e-15         1 9.992007e-15 4.501632e-11         1 2.076923e-08
223  2.590752e-15         1 7.438494e-15  2.96758e-11         1 2.073955e-08
224  3.203268e-15         1 4.218847e-15 4.462978e-11         1 2.069492e-08
226  2.513132e-15         1 1.665335e-15 2.380553e-11         1  2.02578e-08
228  8.000274e-16         1 8.881784e-16 2.928074e-11         1 2.020762e-08
230  6.667216e-16         1 2.220446e-16 2.644072e-09         1 1.754439e-08
225           NaN       NaN          NaN 4.133196e-10         1  2.02816e-08
227           NaN       NaN          NaN 2.090073e-11         1  2.02369e-08
229           NaN       NaN          NaN 1.915667e-11         1 2.018846e-08
231           NaN       NaN          NaN 8.206594e-12         1 1.753618e-08
232           NaN       NaN          NaN 2.810019e-11         1 1.750808e-08
233           NaN       NaN          NaN 1.469202e-12         1 1.750661e-08
234           NaN       NaN          NaN 1.791823e-11         1 1.748869e-08
235           NaN       NaN          NaN 7.914547e-09         1 9.574146e-09
236           NaN       NaN          NaN 9.459713e-12         1 9.564686e-09
237           NaN       NaN          NaN 8.997999e-12         1 9.555688e-09
238           NaN       NaN          NaN 2.312338e-12         1 9.553376e-09
239           NaN       NaN          NaN 2.852462e-12         1 9.550524e-09
240           NaN       NaN          NaN 9.002972e-09         1 5.475517e-10
241           NaN       NaN          NaN 2.816684e-12         1  5.44735e-10
242           NaN       NaN          NaN 1.639808e-12         1 5.430952e-10
243           NaN       NaN          NaN 1.558068e-12         1 5.415371e-10
244           NaN       NaN          NaN 1.794712e-12         1 5.397425e-10
245           NaN       NaN          NaN 4.569197e-12         1 5.351732e-10
246           NaN       NaN          NaN 3.991432e-12         1 5.311818e-10
247           NaN       NaN          NaN  1.48429e-12         1 5.296975e-10
248           NaN       NaN          NaN  9.45804e-12         1 5.202395e-10
249           NaN       NaN          NaN 7.367474e-12         1 5.128721e-10
250           NaN       NaN          NaN 1.533775e-11         1 4.975343e-10
251           NaN       NaN          NaN 1.288323e-11         1 4.846511e-10
252           NaN       NaN          NaN 4.119826e-12         1 4.805313e-10
253           NaN       NaN          NaN 2.848955e-11         1 4.520417e-10
254           NaN       NaN          NaN 2.153303e-11         1 4.305087e-10
```

(continues on next page)

(continued from previous page)

```
255           NaN       NaN          NaN 2.401756e-11         1 4.064912e-10
256           NaN       NaN          NaN 2.364939e-11         1 3.828418e-10
257           NaN       NaN          NaN 4.549044e-12         1 3.782927e-10
258           NaN       NaN          NaN 3.505759e-11         1 3.432351e-10
259           NaN       NaN          NaN 4.316564e-11         1 3.000695e-10
260           NaN       NaN          NaN 1.428383e-11         1 2.857857e-10
261           NaN       NaN          NaN 3.440264e-11         1 2.513831e-10
262           NaN       NaN          NaN 4.775503e-14         1 2.513354e-10
263           NaN       NaN          NaN 9.695625e-12         1 2.416398e-10
264           NaN       NaN          NaN 4.360835e-11         1 1.980315e-10
265           NaN       NaN          NaN 8.192855e-12         1 1.898386e-10
266           NaN       NaN          NaN 4.524329e-11         1 1.445953e-10
267           NaN       NaN          NaN 5.492698e-14         1 1.445404e-10
268           NaN       NaN          NaN  1.10606e-11         1 1.334798e-10
269           NaN       NaN          NaN 9.141233e-14         1 1.333884e-10
270           NaN       NaN          NaN 9.353046e-12         1 1.240353e-10
271           NaN       NaN          NaN 1.118703e-11         1 1.128483e-10
272           NaN       NaN          NaN 1.160805e-13         1 1.127322e-10
273           NaN       NaN          NaN 1.367965e-13         1 1.125954e-10
274           NaN       NaN          NaN 1.828934e-13         1 1.124125e-10
275           NaN       NaN          NaN 1.578676e-13         1 1.122547e-10
276           NaN       NaN          NaN  1.74829e-13         1 1.120798e-10
277           NaN       NaN          NaN 2.204896e-13         1 1.118593e-10
278           NaN       NaN          NaN 1.607224e-13         1 1.116985e-10
279           NaN       NaN          NaN 2.648382e-13         1 1.114338e-10
280           NaN       NaN          NaN  6.39425e-13         1 1.107944e-10
281           NaN       NaN          NaN 1.809595e-13         1 1.106134e-10
282           NaN       NaN          NaN 2.707045e-13         1 1.103427e-10
283           NaN       NaN          NaN 1.970095e-13         1 1.101457e-10
284           NaN       NaN          NaN 2.664828e-13         1 1.098792e-10
285           NaN       NaN          NaN 4.382094e-12         1 1.054972e-10
286           NaN       NaN          NaN 1.397449e-13         1 1.053574e-10
287           NaN       NaN          NaN 1.477025e-13         1 1.052097e-10
288           NaN       NaN          NaN 1.615116e-13         1 1.050482e-10
289           NaN       NaN          NaN 1.259848e-13         1 1.049222e-10
290           NaN       NaN          NaN 1.916858e-11         1 8.575363e-11
291           NaN       NaN          NaN 5.610997e-14         1 8.569756e-11
292           NaN       NaN          NaN 1.640735e-13         1 8.553347e-11
293           NaN       NaN          NaN 1.077555e-14         1  8.55227e-11
294           NaN       NaN          NaN 1.011079e-13         1 8.542156e-11
295           NaN       NaN          NaN 4.317568e-11         1 4.224587e-11
296           NaN       NaN          NaN 5.481826e-14         1 4.219103e-11
297           NaN       NaN          NaN 5.099189e-14         1 4.214007e-11
298           NaN       NaN          NaN 1.407973e-14         1 4.212597e-11
299           NaN       NaN          NaN 1.710537e-14         1 4.210887e-11
300           NaN       NaN          NaN 3.929458e-11         1 2.814304e-12
301           NaN       NaN          NaN 1.865038e-14         1 2.795653e-12
302           NaN       NaN          NaN 1.020398e-14         1 2.785439e-12
303           NaN       NaN          NaN  1.61517e-14         1  2.76934e-12
304           NaN       NaN          NaN 1.513325e-14         1 2.754241e-12
305           NaN       NaN          NaN 3.490552e-14         1  2.71938e-12
306           NaN       NaN          NaN 3.002566e-14         1 2.689404e-12
307           NaN       NaN          NaN 1.066545e-14         1 2.678746e-12
308           NaN       NaN          NaN 6.886354e-14         1 2.609912e-12
309           NaN       NaN          NaN 5.300219e-14         1 2.556955e-12
310           NaN       NaN          NaN  9.01918e-14         1 2.466805e-12
311           NaN       NaN          NaN 7.905782e-14         1 2.387757e-12
312           NaN       NaN          NaN 2.253344e-14         1 2.365219e-12
313           NaN       NaN          NaN 1.580737e-13         1 2.207123e-12
314           NaN       NaN          NaN 1.355182e-13         1 2.071565e-12
315           NaN       NaN          NaN 1.186181e-13         1 1.952993e-12
```

(continues on next page)

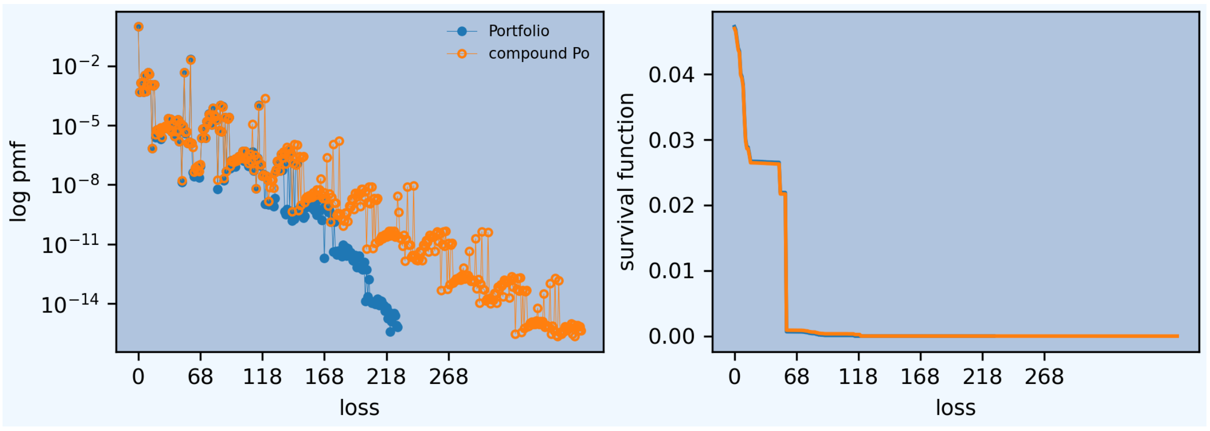
(continued from previous page)

```
316           NaN       NaN          NaN 1.363317e-13        1 1.816658e-12
317           NaN       NaN          NaN 1.995775e-14        1 1.796674e-12
318           NaN       NaN          NaN 1.624001e-13        1 1.634248e-12
319           NaN       NaN          NaN  2.30203e-13        1 1.404099e-12
320           NaN       NaN          NaN 7.025782e-14        1 1.333822e-12
321           NaN       NaN          NaN 1.935323e-13        1  1.14031e-12
322           NaN       NaN          NaN 3.044294e-16        1 1.139977e-12
323           NaN       NaN          NaN 5.294786e-14        1 1.087019e-12
324           NaN       NaN          NaN 1.904147e-13        1 8.966161e-13
325           NaN       NaN          NaN  4.47376e-14        1 8.518741e-13
326           NaN       NaN          NaN 2.081715e-13        1 6.437073e-13
327           NaN       NaN          NaN 3.651783e-16        1 6.433742e-13
328           NaN       NaN          NaN 4.841219e-14        1 5.949685e-13
329           NaN       NaN          NaN 5.792017e-16        1 5.944134e-13
330           NaN       NaN          NaN 4.096891e-14        1 5.534462e-13
331           NaN       NaN          NaN 4.899058e-14        1 5.044853e-13
332           NaN       NaN          NaN 7.188962e-16        1 5.038192e-13
333           NaN       NaN          NaN 7.541253e-16        1 5.030421e-13
334           NaN       NaN          NaN 1.051268e-15        1 5.020429e-13
335           NaN       NaN          NaN 1.215004e-15        1 5.008216e-13
336           NaN       NaN          NaN 9.354349e-16        1 4.999334e-13
337           NaN       NaN          NaN 1.222217e-15        1 4.987122e-13
338           NaN       NaN          NaN 8.942888e-16        1  4.97824e-13
339           NaN       NaN          NaN 1.412135e-15        1 4.963807e-13
340           NaN       NaN          NaN 5.994606e-15        1 4.903855e-13
341           NaN       NaN          NaN 9.247895e-16        1 4.894973e-13
342           NaN       NaN          NaN 1.322465e-15        1 4.881651e-13
343           NaN       NaN          NaN 1.014759e-15        1 4.871659e-13
344           NaN       NaN          NaN 1.290385e-15        1 4.858336e-13
345           NaN       NaN          NaN 3.144116e-14        1 4.544143e-13
346           NaN       NaN          NaN 6.647595e-16        1 4.537482e-13
347           NaN       NaN          NaN 8.041342e-16        1  4.52971e-13
348           NaN       NaN          NaN 7.163302e-16        1 4.523049e-13
349           NaN       NaN          NaN 6.480017e-16        1 4.516387e-13
350           NaN       NaN          NaN 1.043598e-13        1 3.472778e-13
351           NaN       NaN          NaN 2.970056e-16        1 3.469447e-13
352           NaN       NaN          NaN 7.695338e-16        1 3.461675e-13
354           NaN       NaN          NaN 4.549113e-16        1 3.457234e-13
355           NaN       NaN          NaN  1.88428e-13        1 1.573186e-13
356           NaN       NaN          NaN 2.541842e-16        1 1.570966e-13
357           NaN       NaN          NaN 2.321641e-16        1 1.568745e-13
360           NaN       NaN          NaN   1.4293e-13        1 1.398881e-14
368           NaN       NaN          NaN 3.653724e-16        1 1.365574e-14
369           NaN       NaN          NaN 3.012033e-16        1 1.332268e-14
370           NaN       NaN          NaN 4.262703e-16        1 1.287859e-14
371           NaN       NaN          NaN 3.993759e-16        1  1.24345e-14
373           NaN       NaN          NaN  7.04541e-16        1 1.176836e-14
374           NaN       NaN          NaN 6.775864e-16        1 1.110223e-14
375           NaN       NaN          NaN 4.868756e-16        1 1.065814e-14
376           NaN       NaN          NaN  6.44266e-16        1 9.992007e-15
378           NaN       NaN          NaN 6.361091e-16        1 9.325873e-15
379           NaN       NaN          NaN 9.883173e-16        1 8.326673e-15
380           NaN       NaN          NaN 2.902703e-16        1 7.993606e-15
381           NaN       NaN          NaN 8.696954e-16        1 7.105427e-15
383           NaN       NaN          NaN  2.30469e-16        1 6.883383e-15
384           NaN       NaN          NaN 7.177201e-16        1 6.217249e-15
386           NaN       NaN          NaN 7.941167e-16        1 5.440093e-15
410           NaN       NaN          NaN 4.545908e-16        1 4.996004e-15
415           NaN       NaN          NaN 6.831854e-16        1  4.32987e-15
420           NaN       NaN          NaN 4.481143e-16        1 3.885781e-15
```

(continues on next page)

(continued from previous page)

```
In [140]: fig, axs = plt.subplots(1,2, figsize=(3.5*2, 2.45), constrained_
↪layout=True, squeeze=True)

In [141]: ax0, ax1 = axs.flat

In [142]: bit[('exact', 'p_total')].plot(marker='.', lw=.25, logy=True, ax=ax0,␣
↪label='Portfolio');

In [143]: bit[('comp Po', 'p_total')].plot(marker='.', markerfacecolor='None', lw=.
↪25, logy=True, ax=ax0, label='compound Po');

In [144]:  (1-bit[('exact', 'p_total')].cumsum()).plot(ax=ax1);

In [145]:  (1-bit[('comp Po', 'p_total')].cumsum()).plot(ax=ax1);

In [146]: ax0.legend();

In [147]: ax0.set(ylabel='log pmf');

In [148]: ax1.set(ylabel='survival function');
```



Exercise 9.73.

An insurance company sold one-year term life insurance on a group of 2,300 independent lives as given in the next table.

	Benefit	q	Number
Class			
1	100, 000	0.1	500
2	200, 000	0.02	500
3	300, 000	0.02	500
4	200, 000	0.1	300
5	200, 000	0.1	500

The insurance company reinsures amounts in excess of 100,000 on each life. The reinsurer wishes to charge a premium that is sufficient to guarantee that it will lose money 5% of the time on such groups. Obtain the appropriate premium by each of the following ways:

1. Using a normal approximation to the aggregate claims distribution.
2. Using a lognormal approximation.
3. Using a gamma approximation.
4. Using the compound Poisson approximation that matches the means.

In order to make the answer self-contained, the code below includes the data munging to re-create the table, pasted from a pdf.

```
In [149]: data = '''1
   .....: 100,000
   .....: 0.10
   .....: 500
   .....: 2
   .....: 200,000
   .....: 0.02
   .....: 500
   .....: 3
   .....: 300,000
   .....: 0.02
   .....: 500
   .....: 4
   .....: 200,000
   .....: 0.10
   .....: 300
   .....: 5
   .....: 200,000
   .....: 0.10
   .....: 500'''
   .....:

In [150]: sdata = data.split('\n')

In [151]: df = pd.DataFrame(zip(*[iter(sdata)]*4),
   .....:                   columns=['Class', 'Benefit', 'q', 'Number'])
   .....:

In [152]: df.Benefit = df.Benefit.str.replace(',','').astype(float)

In [153]: df.q = df.q.astype(float)

In [154]: df.Number = df.Number.astype(int)

In [155]: df = df.set_index('Class')

In [156]: qd(df)

       Benefit     q  Number
Class
1        1e+05   0.1     500
2        2e+05  0.02     500
3        3e+05  0.02     500
4        2e+05   0.1     300
5        2e+05   0.1     500
```

Next, build the exact solution for the gross book as a `Portfolio` (extra credit).

```
In [157]: a = [build(f'agg Class.{i} {r.q * r.Number} claims '
   .....:            f'dsev [{r.Benefit / 100000}] '
   .....:            f'binomial {r.q}')
   .....:          for i, r in df.iterrows()]
   .....:

In [158]: p = Portfolio('KPW.9.73p', a)

In [159]: p.update(log2=10, bs=1, remove_fuzz=True)

In [160]: qd(p)

               E[X]  Est E[X]     Err E[X]     CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
unit    X
```

(continues on next page)

(continued from previous page)

```
Class.1 Freq     50                        0.13416           0.11926
        Sev       1        1         0           0         0
        Agg      50       50 -8.4155e-14   0.13416   0.13416  0.11926     0.11926
Class.2 Freq     10                        0.31305           0.30666
        Sev       2        2         0           0         0
        Agg      20       20  5.7554e-13   0.31305   0.31305  0.30666     0.30666
Class.3 Freq     10                        0.31305           0.30666
        Sev       3        3         0           0         0
        Agg      30       30  2.9909e-13   0.31305   0.31305  0.30666     0.30666
Class.4 Freq     30                        0.17321           0.15396
        Sev       2        2         0           0         0
        Agg      60       60 -1.3123e-13   0.17321   0.17321  0.15396     0.15396
Class.5 Freq     50                        0.13416           0.11926
        Sev       2        2         0           0         0
        Agg     100      100 -1.1879e-13   0.13416   0.13416  0.11926     0.11926
total   Freq    150                       0.077917          0.070413
        Sev  1.7333   1.7333         0   0.33086          0.081688
        Agg     260      260    9.77e-15 0.082527  0.082527 0.083622    0.083622
log2 = 10, bandwidth = 1, validation: not unreasonable.
```

Build the reinsurer's loss distribution exactly, as `p_ceded`, a `Portfolio`, and the compound Poisson approximation `cp_ceded`, an `Aggregate`. The temporary variable `bit` is used to calculate the mixed severity distribution.

```
In [161]: a_ceded = [build(f'agg Class.{i}.c {r.q * r.Number} claims '
   .....:              f'dsev [{r.Benefit / 100000 - 1}] '
   .....:              f'binomial {r.q}')
   .....:            for i, r in df.query('Benefit > 100000').iterrows()]
   .....:

In [162]: p_ceded = Portfolio('KPW.9.73pc', a_ceded)

In [163]: p_ceded.update(log2=10, bs=1, remove_fuzz=True)

In [164]: qd(p_ceded)

                E[X] Est E[X]    Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit       X
Class.2.c Freq    10                        0.31305           0.30666
          Sev      1        1         0           0         0
          Agg     10       10  1.0121e-12   0.31305   0.31305  0.30666     0.30666
Class.3.c Freq    10                        0.31305           0.30666
          Sev      2        2         0           0         0
          Agg     20       20  5.7554e-13   0.31305   0.31305  0.30666     0.30666
Class.4.c Freq    30                        0.17321           0.15396
          Sev      1        1         0           0         0
          Agg     30       30 -9.6367e-14   0.17321   0.17321  0.15396     0.15396
Class.5.c Freq    50                        0.13416           0.11926
          Sev      1        1         0           0         0
          Agg     50       50 -8.4155e-14   0.13416   0.13416  0.11926     0.11926
total     Freq   100                       0.095708          0.087165
          Sev    1.1      1.1         0   0.27273            2.6667
          Agg    110      110  1.4877e-14       0.1       0.1  0.10689     0.10689
log2 = 10, bandwidth = 1, validation: not unreasonable.

In [165]: bit = df.query('Benefit > 100000')

In [166]: bit['Claims'] = bit.q * bit.Number

In [167]: bit.groupby('Benefit').Claims.sum()
Out[167]:
Benefit
```

(continues on next page)

(continued from previous page)

```
200000.0    90.0
300000.0    10.0
Name: Claims, dtype: float64

In [168]: cp_ceded = build('agg CP.Approx '
   .....:                  f'{bit.Claims.sum()} claims '
   .....:                  f'dsev [1 2] [0.9 0.1] '
   .....:                  'poisson')
   .....:

In [169]: qd(cp_ceded)

      E[X] Est E[X]   Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   100                         0.1                0.1
Sev    1.1      1.1          0 0.27273   0.27273   2.6667      2.6667
Agg    110      110 6.1506e-14 0.10365   0.10365  0.11469     0.11469
log2 = 14, bandwidth = 1, validation: not unreasonable.
```

Compute the various estimated premiums, the 95%iles of the aggregate loss distribution.

```
In [170]: prem_confidence = 0.95

In [171]: appx = p_ceded.approximate('all')

In [172]: ans = {k: v.ppf(prem_confidence) for k, v in appx.items()}

In [173]: ans['FFT'] = p_ceded.q(prem_confidence)

In [174]: ans['Comp Po'] = cp_ceded.q(prem_confidence)

In [175]: qd(pd.DataFrame(ans.values(),
   .....:                 index=pd.Index(ans.keys(), name='method'),
   .....:                 columns=['premium']).sort_values('premium'),
   .....:     accuracy=5)
   .....:

          premium
method
FFT           128
norm       128.09
slognorm   128.42
sgamma     128.42
gamma       128.7
lognorm    128.97
Comp Po       129
```

Exercise 9.74. A group insurance contract covers 1,000 employees. An employee can have at most one claim per year. For 500 employees, there is a 0.02 probability of a claim, and when there is a claim, the amount has an exponential distribution with mean 500. For 250 other employees, there is a 0.03 probability of a claim and amounts are exponential with mean 750. For the remaining 250 employees, the probability is 0.04 and the mean is 1,000. Determine the exact mean and variance of total claims payments. Next, construct a compound Poisson model with the same mean and determine the variance of this model.

```
In [176]: kpw_9_74p = build('port KPW.9.74p '
   .....:                   'agg A 10. claims sev  500 * expon binomial 0.02 '
   .....:                   'agg B 7.5 claims sev  750 * expon binomial 0.03 '
   .....:                   'agg C 10. claims sev 1000 * expon binomial 0.04 ')
   .....:

In [177]: qd(kpw_9_74p)
```

(continues on next page)

(continued from previous page)

```
            E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
unit  X
A     Freq    10                        0.31305             0.30666
      Sev    500       500 -6.6667e-07        1         1         2           2
      Agg   5000      5000 -6.6667e-07 0.44497   0.44497   0.66748     0.66748
B     Freq   7.5                        0.35963             0.34851
      Sev    750       750  -2.963e-07        1         1         2           2
      Agg   5625      5625  -2.963e-07 0.51251   0.51251   0.76882     0.76882
C     Freq    10                        0.30984             0.29693
      Sev   1000      1000 -1.6667e-07        1         1         2           2
      Agg  10000     10000 -1.6667e-07 0.44272   0.44272   0.66417     0.66417
total Freq  27.5                        0.18781             0.18203
      Sev    750       750 -3.2323e-07  1.0778                2.372
      Agg  20625     20625 -3.2323e-07 0.27794   0.27794   0.44276     0.44276
log2 = 16, bandwidth = 2, validation: not unreasonable.

In [178]: mv(kpw_9_74p)
mean     = 20625
variance = 3.286094e+07
std dev  = 5732.45
```

Compound Poisson approximation is easy to construct as a mixture.

```
In [179]: kpw_9_74cp = build('agg KPW.9.74.cp [10 7.5 10] claims sev [500 750␣
↪1000] * expon poisson')

In [180]: qd(kpw_9_74cp)

      E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)   Skew(X) Est Skew(X)
X
Freq  27.5                        0.19069             0.19069
Sev    750       750 -3.2323e-07  1.0778    1.0778     2.372       2.372
Agg  20625     20625 -3.2323e-07 0.28036   0.28036   0.44729     0.44729
log2 = 16, bandwidth = 2, validation: not unreasonable.

In [181]: mv(kpw_9_74cp)
mean     = 20625
variance = 3.34375e+07
std dev  = 5782.52
```

2.13.6 Bahnemann Monograph

Examples from the CAS published monograph Bahnemann [2015], Distributions for actuaries, which is a text for CAS Part 8. It is available for free on the [CAS Website](https://www.casact.org/monograph/cas-monograph-no-2).

Contents

Simple Discrete Aggregate, Example 4.1

Assume that n = 0, 1, 2 are the only possible numbers of claims and they occur with probabilities 0.6, 0.3 and 0.1, and that there exist just three potential claim sizes: 1, 2, and 3 with probabilities 0.4, 0.5 and 0.1. (Note: the text uses claim sizes 100, 200 and 300.) Compute the distribution of possible outcomes and its mean and variance.

Imports and convenience functions.

```
In [1]: from aggregate import build, qd, mv

In [2]: import matplotlib.pyplot as plt
```

Build the aggregate and display key statistics.

```
In [3]: a = build('agg Bahn.4.1 dfreq[0 1 2][.6 .3 .1] '
   ...:           'dsev[1 2 3][.4 .5 .1]')
   ...:

In [4]: qd(a)

      E[X] Est E[X]     Err E[X]    CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   0.5                        1.3416            0.99381
Sev    1.7       1.7 -1.1102e-16 0.37665   0.37665  0.36568     0.36568
Agg   0.85      0.85 -2.2204e-16  1.4435    1.4435   1.3333      1.3333
log2 = 4, bandwidth = 1, validation: not unreasonable.

In [5]: mv(a)
mean     = 0.85
variance = 1.5055
std dev  = 1.22699
```

Display all possible outcomes. Compare with the table on p. 107.

```
In [6]: qd(a.density_df.query('p_total > 0') [['p_total', 'F']])

      p_total     F
loss
0.0       0.6   0.6
1.0      0.12  0.72
2.0     0.166 0.886
3.0      0.07 0.956
4.0     0.033 0.989
5.0      0.01 0.999
6.0     0.001     1
```

Poisson-Gamma (Tweedie) Aggregate, Example 4.2

The text considers a Tweedie with expected claim count $\lambda = 2.5$ and gammma shape 3 and scale 400. It computes the mean, variance and skewness, and uses the series expansion for the distribution to compute the CDF at various points (Table 4.1). These results can be replicated as follows.

```
In [7]: a = build('agg Bahn.4.2 2.5 claims '
   ...:           'sev 400 * gamma 3 poisson')
   ...:

In [8]: qd(a)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   2.5                      0.63246            0.63246
Sev   1200     1200  1.5543e-15 0.57735   0.57735   1.1547      1.1547
Agg   3000     3000 -2.7578e-13  0.7303    0.7303  0.91287     0.91287
log2 = 16, bandwidth = 1/2, validation: not unreasonable.

In [9]: mv(a)
mean     = 3000
variance = 4800000
std dev  = 2190.89
```

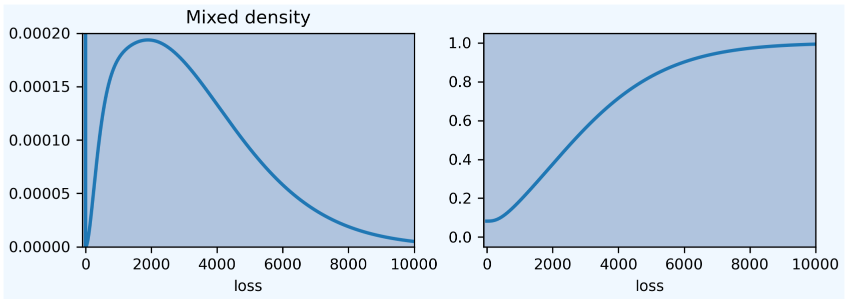
Extract various points of the pmf, cdf, and sf. The adjustment to the index is cosmetic. `aggregate` returns the entire distribution. The left plot shows the mixed density, with a mass at zero; right shows the cdf.

```
In [10]: bit = a.density_df.loc[
   ....:     sorted(np.hstack((500, np.arange(0, 10000.5, 1000)))),
   ....:     ['p', 'F', 'S']]
   ....:

In [11]: qd(bit, accuracy=4)

                 p        F        S
loss
0.0       0.082085 0.082085  0.91792
500.0   5.9764e-05  0.10958  0.89042
1000.0  8.8058e-05  0.18677  0.81323
2000.0  9.6726e-05  0.37558  0.62442
3000.0  8.6514e-05  0.56132  0.43868
4000.0  6.6611e-05  0.71519  0.28481
5000.0  4.5895e-05  0.82731  0.17269
6000.0  2.8971e-05  0.90135 0.098646
7000.0  1.7022e-05  0.94653 0.053468
8000.0  9.4148e-06  0.97234 0.027662
9000.0  4.9432e-06  0.98627 0.013727
10000.0 2.4799e-06  0.99344 0.006561

In [12]: fig, axs = plt.subplots(1, 2, figsize=(3.5*2, 2.45), constrained_
↪layout=True, squeeze=True)

In [13]: ax0, ax1 = axs.flat

In [14]: (a.density_df.p / a.bs).plot(ylim=[0, 0.0002], xlim=[-100, 10000], lw=2,␣
↪ax=ax0)
Out[14]: <Axes: xlabel='loss'>

In [15]: ax0.set(title='Density')
Out[15]: [Text(0.5, 1.0, 'Density')]

In [16]: a.density_df.F.plot(ylim=[-0.05, 1.05], xlim=[-100, 10000], lw=2, ax=ax1)
```

(continues on next page)

(continued from previous page)

```
Out[16]: <Axes: xlabel='loss'>

In [17]: ax0.set(title='Mixed density');

In [18]: ax1.set(title='Distribution function');
```



Approximations to the Tweedie, Example 4.3

`aggregate` largely circumvents the need for approximations, but it does support their creation. The following reproduces Table 4.3.

```
In [19]: fz = a.approximate('all')

In [20]: bit['Normal'] = fz['norm'].cdf(bit.index)

In [21]: bit['Norm err'] = bit.Normal / bit.F - 1

In [22]: bit['sGamma'] = fz['sgamma'].cdf(bit.index)

In [23]: bit['sGamma err'] = bit.sGamma / bit.F - 1

In [24]: qd(bit, accuracy=4)

                 p        F        S   Normal   Norm err   sGamma   sGamma err
loss
0.0       0.082085 0.082085  0.91792 0.085452   0.041016 0.045932     -0.44044
500.0   5.9764e-05  0.10958  0.89042  0.12692    0.15821  0.10104    -0.077947
1000.0  8.8058e-05  0.18677  0.81323  0.18066  -0.032733  0.17751    -0.049594
2000.0  9.6726e-05  0.37558  0.62442  0.32404   -0.13723  0.36803    -0.020087
3000.0  8.6514e-05  0.56132  0.43868      0.5   -0.10924  0.56073   -0.0010453
4000.0  6.6611e-05  0.71519  0.28481  0.67596  -0.054848  0.71854    0.0046875
5000.0  4.5895e-05  0.82731  0.17269  0.81934 -0.0096235  0.83105    0.0045249
6000.0  2.8971e-05  0.90135 0.098646  0.91455   0.014639  0.90379    0.0027037
7000.0  1.7022e-05  0.94653 0.053468  0.96606   0.020626  0.94752    0.0010466
8000.0  9.4148e-06  0.97234 0.027662  0.98876    0.01689  0.97238   4.3133e-05
9000.0  4.9432e-06  0.98627 0.013727  0.99692    0.01079  0.98589 -0.00038586
10000.0 2.4799e-06  0.99344 0.006561   0.9993  0.0059007  0.99298 -0.00046669
```

Here is Table 4.4. The FFT overstates $F(0)$ because of discretization error.

```
In [25]: a2 = build('agg Bahn.4.2b 10 claims '
   ....:            'sev 6000 * gamma 0.05 poisson')
   ....:

In [26]: qd(a2)

      E[X]  Est E[X]    Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq    10                         0.31623             0.31623
```

(continues on next page)

(continued from previous page)

```
Sev    300   299.94 -0.00018672  4.4721     4.473   8.9443      8.9441
Agg   3000   2999.4 -0.00018672  1.4491    1.4494   2.8293      2.8293
log2 = 16, bandwidth = 5, validation: fails sev mean, agg mean.

In [27]: fz = a2.approximate('all')

In [28]: bit = a2.density_df.loc[
   ....:     sorted(np.hstack((500, np.arange(0, 20000, 2000)))),
   ....:     ['p', 'F', 'S']]
   ....:

In [29]: bit['Normal'] = fz['norm'].cdf(bit.index)

In [30]: bit['Norm err'] = bit.Normal / bit.F - 1

In [31]: bit['sGamma'] = fz['sgamma'].cdf(bit.index)

In [32]: bit['sGamma err'] = bit.sGamma / bit.F - 1

In [33]: qd(bit, accuracy=4)

                 p        F        S  Normal  Norm err  sGamma  sGamma err
loss
0.0       0.047861 0.047861  0.95214 0.24512    4.1215 0.12322      1.5746
500.0    0.0014101  0.33964  0.66036 0.28267  -0.16774 0.33446   -0.015242
2000.0  0.00055391  0.59253  0.40747 0.40909  -0.30959  0.5887  -0.0064723
4000.0  0.00028651  0.75148  0.24852 0.59101  -0.21353 0.75043  -0.0013915
6000.0  0.00017056  0.84025  0.15975 0.75496   -0.1015 0.84022 -3.1891e-05
8000.0  0.00010741  0.89466  0.10534 0.87498 -0.022001 0.89493  0.00030275
10000.0 6.9725e-05  0.92946 0.070541 0.94633  0.018153 0.92976   0.0003232
12000.0 4.6131e-05  0.95227 0.047735 0.98079  0.029954 0.95251  0.00025575
14000.0 3.0919e-05  0.96745 0.032549  0.9943  0.027755 0.96762  0.00017682
16000.0 2.0919e-05  0.97768 0.022323 0.99861  0.021408 0.97779  0.00011036
18000.0 1.4255e-05  0.98462  0.01538 0.99972  0.015336 0.98468  6.1254e-05
```

Poisson-Discrete Distribution, Example 4.4

The claim-count random variable is Poisson distributed with mean 1.75. Severity has a discrete distribution with outcomes 1, 2, 3, 4, 5 occurring with probabilities 0.2, 0.4, 0.2, 0.15, 0.05 respectively. Compute the aggregate distribution.

Here is Table 4.5.

```
In [34]: a = build('agg Bahn.4.4 1.75 claims '
   ....:           'dsev [1 2 3 4 5] [.2 .4 .2 .15 .05] '
   ....:           'poisson')
   ....:

In [35]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq   1.75                      0.75593            0.75593
Sev    2.45     2.45 -2.2204e-16 0.45588   0.45588  0.55603     0.55603
Agg  4.2875   4.2875  2.2204e-16 0.83078   0.83078  0.95453     0.95453
log2 = 7, bandwidth = 1, validation: not unreasonable.

In [36]: qd(a.density_df.query('p > .001')[['p', 'F', 'S']], accuracy=4)

             p       F         S
```

(continues on next page)

(continued from previous page)

```
loss
0.0    0.17377 0.17377   0.82623
1.0   0.060821 0.23459   0.76541
2.0    0.13229 0.36688   0.63312
3.0    0.10464 0.47152   0.52848
4.0    0.11704 0.58855   0.41145
5.0   0.093249  0.6818    0.3182
6.0   0.078644 0.76045   0.23955
7.0   0.064101 0.82455   0.17545
8.0   0.049889 0.87444   0.12556
9.0   0.037655 0.91209  0.087908
10.0   0.02737 0.93946  0.060537
11.0  0.019672 0.95913  0.040865
12.0  0.013764  0.9729  0.027101
13.0 0.0094201 0.98232  0.017681
14.0 0.0063168 0.98864  0.011364
15.0 0.0041654  0.9928 0.0071986
16.0 0.0027032  0.9955 0.0044954
17.0 0.0017249 0.99723 0.0027705
18.0 0.0010842 0.99831 0.0016863
```

Poisson-Gamma Distribution, Example 4.5

Aggregate losses have Poisson frequency with mean 2.5 and gamma severity with shape 3 and scale 400. Hence the aggregate mean equals 1,200 and variance equals 480,000. Now approximate the distribution function using FFT with a fine bucket size and the midpoint method for assigning claim-size probabilities and then `bs=20` and `bs=100`.

Here is Table 4.6, comparing the distributions. The `update` method re-runs the FFT computation with different options, here altering `bs`.

```
In [37]: import numpy as np

In [38]: import pandas as pd

In [39]: a = build('agg Bahn.4.5 2.5 claims '
   ....:           'sev 400 * gamma 3 poisson')
   ....:

In [40]: qd(a)

      E[X]  Est E[X]    Err E[X]   CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
X
Freq   2.5                       0.63246            0.63246
Sev   1200     1200  1.5543e-15 0.57735    0.57735    1.1547       1.1547
Agg   3000     3000 -2.7578e-13  0.7303     0.7303   0.91287      0.91287
log2 = 16, bandwidth = 1/2, validation: not unreasonable.

In [41]: xs = sorted(np.hstack((500, np.arange(0, 10001, 1000))))

In [42]: bit = a.density_df.loc[xs, ['F']]

In [43]: a.update(bs=100)

In [44]: bit100 = a.density_df.loc[xs,  ['F']]

In [45]: a.update(bs=20)

In [46]: bit20 = a.density_df.loc[xs,  ['F']]

In [47]: bit = pd.concat((bit, bit100, bit20), axis=1, keys=['h0.25', 'h100', 'h20
```

(continues on next page)

(continued from previous page)

```
↪'])

In [48]: bit[('h100', 'Rel Error')] = bit[('h100', 'F')] / bit[('h0.25', 'F')] - 1

In [49]: bit[('h20', 'Rel Error')] = bit[('h20', 'F')] / bit[('h0.25', 'F')] - 1

In [50]: bit = bit.sort_index(axis=1)

In [51]: qd(bit, accuracy=4)

           h0.25     h100                   h20
               F        F  Rel Error          F  Rel Error
loss
0.0     0.082085 0.082146 0.00074147 0.082086 6.3895e-06
500.0    0.10958  0.11579   0.056708  0.11076   0.010735
1000.0   0.18677  0.19565   0.047526  0.18849  0.0092186
2000.0   0.37558  0.38525   0.025749  0.37747  0.0050273
3000.0   0.56132  0.56991   0.015294  0.56301   0.003004
4000.0   0.71519  0.72175  0.0091789  0.71648  0.0018127
5000.0   0.82731   0.8318  0.0054318   0.8282  0.0010783
6000.0   0.90135  0.90417  0.0031276  0.90192 0.00062395
7000.0   0.94653  0.94818  0.0017394  0.94686 0.00034868
8000.0   0.97234  0.97324 0.00093101  0.97252  0.0001875
9000.0   0.98627  0.98675 0.00047913  0.98637 9.6942e-05
10000.0  0.99344  0.99367 0.00023728  0.99349 4.8226e-05
```

Poisson-Lognormal Distribution With Limit, Example 4.15

Consider an aggregate distribution with mean 3 Poisson frequency and lognormal claim size with parameters $(\mu, \sigma) = (6, 1.5)$. Moreover, claim size is limited by a policy limit of 1,000. Graph the aggregate distribution.

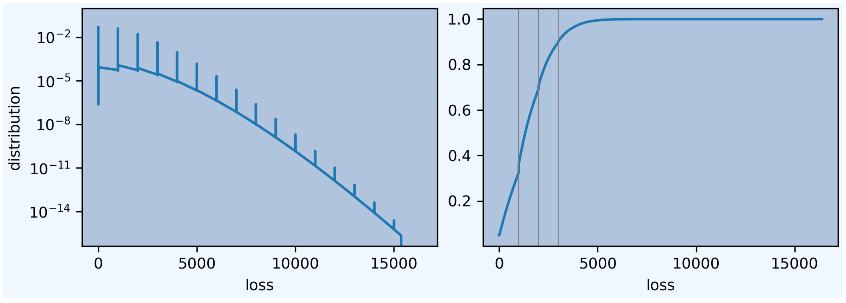
The log density (left) shows the probability masses at outcomes consisting of only limit losses. The distribution (right) shows the corresponding jumps. Compare with Figure 4.4.

```
In [52]: a = build('agg Bahn.4.15 '
   ....:           '3 claims '
   ....:           '1000 xs 0 '
   ....:           'sev exp(6) * lognorm 1.5 '
   ....:           'poisson')
   ....:

In [53]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq      3                      0.57735            0.57735
Sev  502.97   502.97 1.1478e-09 0.74753   0.74753  0.23542     0.23542
Agg  1508.9   1508.9 1.1478e-09 0.72083   0.72083  0.82314     0.82314
log2 = 16, bandwidth = 1/4, validation: not unreasonable.

In [54]: fig, axs = plt.subplots(1, 2, figsize=(2*3.5, 2.45), constrained_
↪layout=True)

In [55]: ax0, ax1 = axs.flat

In [56]: a.density_df.p.plot(ax=ax0, logy=True, label='FFT');

In [57]: a.density_df.F.plot(ax=ax1, label='FFT');

In [58]: ax0.set(ylabel='log density');
```

(continues on next page)

(continued from previous page)

```
In [59]: ax0.set(ylabel='distribution', ylim=[0,1]);

In [60]: ax1.axvline(1000, c='C7', lw=.5);

In [61]: ax1.axvline(2000, c='C7', lw=.5);

In [62]: ax1.axvline(3000, c='C7', lw=.5);
```



Poisson-Gamma Distribution and Approximations, Problems 4.7 and 13

An aggregate distribution has mean 8 Poisson frequency and gamma severity with shape 0.2 and scale 3750. Compute the distribution and compare with normal and shifted-gamma approximations.

```
In [63]: a = build('agg Bahn.4.7 '
   ....:           '8 claims '
   ....:           'sev 3750 * gamma 0.2 '
   ....:           'poisson')
   ....:

In [64]: qd(a)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     8                      0.35355            0.35355
Sev    750   749.98 -2.9119e-05  2.2361    2.2361   4.4721      4.4721
Agg   6000   5999.8 -2.9119e-05 0.86603   0.86605   1.5877      1.5877
log2 = 16, bandwidth = 2, validation: not unreasonable.

In [65]: xs = np.arange(0, 30000,3000)

In [66]: qd(a.density_df.loc[xs, ['p', 'F','S']], accuracy=4)

                 p         F         S
loss
0.0      0.0018002 0.0018002    0.9982
3000.0  0.00020926   0.34209   0.65791
6000.0  0.00014322   0.60704   0.39296
9000.0  8.7227e-05    0.7775    0.2225
12000.0 5.0019e-05   0.87823   0.12177
15000.0 2.7618e-05   0.93494  0.065065
18000.0 1.4853e-05   0.96586  0.034143
21000.0 7.8339e-06   0.98233  0.017666
24000.0 4.0696e-06   0.99096 0.0090364
27000.0 2.0885e-06   0.99542 0.0045787
```

`aggregate` readily computes approximations and returns frozen `scipy.stats` objects.

```
In [67]: fz = a.approximate('all')

In [68]: comp = pd.DataFrame({k: v.cdf(xs) for k, v in fz.items()}, index=xs)

In [69]: comp['agg'] = a.density_df.loc[xs, 'F',]

In [70]: comp.loc[:, [f'{k} err' for k in fz.keys()]] = comp.loc[:, fz.keys()].
↪values / comp.loc[:, ['agg']].values - 1

In [71]: comp = comp.sort_index(axis=1)

In [72]: qd(comp, accuracy=4)

             agg    gamma    gamma err   lognorm   lognorm err     norm    norm err  ␣
↪sgamma   sgamma err   \
0      0.0018002        0           -1         0            -1 0.12411      67.945 0.
↪026306      13.613
3000     0.34209 0.33977  -0.0067746   0.29031     -0.15135 0.28186    -0.17605  0.
↪33624    -0.017089
6000     0.60704 0.61485    0.012868   0.64583     0.063894 0.50001    -0.17631  0.
↪60542   -0.0026701
9000      0.7775 0.78363   0.0078835   0.82019     0.054907 0.71816    -0.07632  0.
↪77824   0.00095764
12000    0.87823 0.88084   0.0029796   0.90331      0.02856   0.8759 -0.0026494  0.
↪87926    0.0011727
15000    0.93494  0.9352  0.00028096   0.94508     0.010852 0.95837    0.025066  0.
↪93559    0.0007034
18000    0.96586 0.96506 -0.00082253   0.96731    0.0015039 0.98954    0.024521  0.
↪96614   0.00028959
21000    0.98233 0.98128  -0.0010704   0.97975   -0.0026265 0.99805    0.016002  0.
↪98239   5.1729e-05
24000    0.99096 0.99002 -0.00095081   0.98703   -0.0039667 0.99973   0.0088504  0.
↪99091  -5.1265e-05
27000    0.99542  0.9947 -0.00072386   0.99145   -0.0039875 0.99997   0.0045731  0.
↪99534   -7.866e-05

       slognorm  slognorm err
0      0.058626        31.567
3000    0.31389     -0.082441
6000    0.59173     -0.025232
9000    0.77906      0.002005
12000   0.88465     0.0073188
15000   0.94022     0.0056501
18000   0.96881     0.0030552
21000    0.9835      0.001191
24000   0.99113     0.0001684
27000   0.99515   -0.00027715
```

Poisson-Lognormal Layer Statistics, Example 5.13

Consider an aggregate distribution with mean 15 Poisson frequency and lognormal claim size with parameters $(\mu, \sigma) = (5.9809, 1.8)$. What are the distribution characteristics for random variable S for claims in the layer 5,000 excess of 3,000?

The exact and FFT-estimated mean, cv, and skewness are reported in the `describe` dataframe, for frequency and severity. The values reported agree with the text, up to rounding.

```
In [73]: a = build('agg Bahn.5.13 '
   ....:           '15 claims 5000 xs 3000 '
   ....:           'sev exp(5.9809) * lognorm 1.8 ! '
```

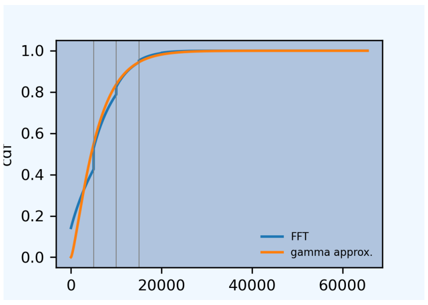
(continues on next page)

(continued from previous page)

```
   ....:        'poisson')
   ....:

In [74]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     15                       0.2582             0.2582
Sev  385.68   385.68 -3.4957e-09  3.1319    3.1319   3.1942      3.1942
Agg  5785.3   5785.3 -3.4982e-09 0.84887   0.84887  0.93405     0.93405
log2 = 16, bandwidth = 1, validation: not unreasonable.

In [75]: mv(a)
mean     = 5785.25
variance = 2.411727e+07
std dev  = 4910.93
```

The exact severity can be accessed directly, as `a.sevs[0].fz`, allowing us to compute the expected layer claim count. The aggregate can then be written in conditional form, producing the same statistics. The distribution function shows probability masses at multiples of the limit.

```
In [76]: xs = 15 * a.sevs[0].fz.sf(3000)

In [77]: print(f'excess claim count = {xs:.5f}')
excess claim count = 1.95359

In [78]: a = build('agg Bahn.5.13b '
   ....:           f'{xs} claims 5000 xs 3000 '
   ....:           'sev exp(5.9809) * lognorm 1.8 '
   ....:           'poisson')
   ....:

In [79]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 1.9536                      0.71546            0.71546
Sev  2961.3   2961.3 -3.4956e-09 0.63853   0.63853 -0.16374    -0.16374
Agg  5785.3   5785.3 -3.4982e-09 0.84887   0.84887  0.93405     0.93405
log2 = 16, bandwidth = 1, validation: not unreasonable.

In [80]: fig, ax = plt.subplots(1,1,figsize=(3.5, 2.45))

In [81]: a.density_df.F.plot(ax=ax, label='FFT');

In [82]: fz = a.approximate('gamma')

In [83]: ax.plot(a.density_df.loss, fz.cdf(a.density_df.loss), c='C1',␣
↪label='gamma approx.');

In [84]: ax.axvline(5000,  c='C7', lw=.5);

In [85]: ax.axvline(10000, c='C7', lw=.5);

In [86]: ax.axvline(15000, c='C7', lw=.5);

In [87]: ax.set(ylabel='cdf');

In [88]: ax.legend(loc='lower right');
```



Lognormal Increased Limits Factors (ILFs), Example 6.3

Indemnity losses for a portfolio of insurance policies have a lognormal claim-size distribution with parameters $(\mu, \sigma) = (7, 2.4)$. The policy per-claim limit applies only to the indemnity portion of a claim, and the average per-claim loss adjustment expense is 2,200. Claim frequency for these policies is 0.0005 per exposure unit, and variable expenses equal 35% of premium.

A lognormal with $\sigma = 2.4$ has cv $\sqrt{\exp(2.4^2) - 1} = 17.78$ and is extremely thick-tailed, despite having moments of all orders. It is challenging to approximate numerically. Luckily, we only need to compute up to 5M. The `aggregate` parameters deliberately select a range that is too narrow for the entire distribution, but adequate for our purposes. Use `log2=17` and select `bs` greater than `5e6 // 2**17 = 38`. We use `bs=50`. It is important to set `normalize=False` to avoid rescaling bucket probabilities to sum to one. These parameters are not a good model for the entire distribution; the mean error is too high.

The `density_df` dataframe includes limited expected values. Here is a sample.

```
In [89]: a = build('agg Bahn.6.3 '
   ....:             '1 claim '
   ....:             'sev exp(7) * lognorm 2.4 '
   ....:             'fixed',
   ....:             bs=50, log2=17,
   ....:             normalize=False,
   ....:           )
   ....:

In [90]: qd(a)

      E[X] Est E[X]   Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq     1                           0
Sev   19536    18328 -0.061835 17.786    7.7935    5680       27.885
Agg   19536    18328 -0.061835 17.786    7.7935    5680       27.885
log2 = 17, bandwidth = 50, validation: fails sev mean, agg mean.

In [91]: xs = [1e5,  5e5, 7.5e5, 1e6, 2e6, 3e6, 4e6, 5e6]

In [92]: qd(a.density_df.loc[xs, ['F', 'S', 'lev']], accuracy=4)

                F           S     lev
loss
100000.0  0.96998    0.030021 8895.8
500000.0  0.99463  0.0053706  13625
750000.0  0.99674  0.0032647  14668
1000000.0 0.99774   0.002257  15345
2000000.0 0.99912 0.00087817  16738
3000000.0 0.99951 0.00048765  17390
4000000.0 0.99968 0.00031609  17782
5000000.0 0.99978 0.00022372  18048
```

The following reproduces Table 6.1. The ILF factors assume fixed (middle) and variable ALAE (right).

```
In [93]: alae = 2200

In [94]: bit = a.density_df.loc[xs, ['lev']]

In [95]: bit['Fixed ALAE'] = (bit.lev + alae) / (bit.lev.iloc[0] + alae)

In [96]: bit['Prop ALAE'] = bit.lev / bit.lev.iloc[0]

In [97]: qd(bit, accuracy=4)

             lev  Fixed ALAE  Prop ALAE
loss
100000.0  8895.8           1          1
500000.0   13625      1.4262     1.5317
750000.0   14668      1.5202     1.6488
1000000.0  15345      1.5812      1.725
2000000.0  16738      1.7067     1.8815
3000000.0  17390      1.7655     1.9549
4000000.0  17782      1.8009     1.9989
5000000.0  18048      1.8248     2.0288
```

Layer Premium, Example 6.4

(Continues *Example 6.3*.) Calculate the premium for successive excess layers of insurance for a policy with exposure equal 400. Use the ILFs under the assumption that the average per-claim ALAE payment is 2,200. Premium amounts for the successive million-dollar layers obtained from these layer factors applied to the basic-limit premium are displayed in Table 6.2 and reproduced below.

```
In [98]: exposure = 400

In [99]: var_exp = 0.35

In [100]: frequency = 0.0005

In [101]: bit['Premium'] = exposure * frequency * (bit['lev'] + alae) / (1 - var_
↪exp)

In [102]: bit['Layer Premium'] = np.diff(bit.Premium, prepend=0)

In [103]: qd(bit)

             lev  Fixed ALAE  Prop ALAE  Premium  Layer Premium
loss
100000.0  8895.8           1          1   3414.1         3414.1
500000.0   13625      1.4262     1.5317   4869.3         1455.2
750000.0   14668      1.5202     1.6488   5190.1         320.73
1000000.0  15345      1.5812      1.725   5398.4         208.38
2000000.0  16738      1.7067     1.8815     5827         428.52
3000000.0  17390      1.7655     1.9549   6027.7         200.71
4000000.0  17782      1.8009     1.9989   6148.3         120.63
5000000.0  18048      1.8248     2.0288   6230.1          81.76
```

Risk Loads, Example 6.5

Example 6.5, computes risk loads as a percentage of standard deviation. `aggregate` can compute multiple limits at once, and the `report_df` dataframe returns individual severity and aggregate distribution statistics. The risk loads can be deduced from these. The risk load can be computed as `k' * ex2` or `k * agg_cv` (not shown).

The following code reproduces Table 6.3. First, define the controlling variables, and then set up the tower of limits within one object, using *Vectorization: Limit Profiles and Mixed Severity*.

```
In [104]: k_prime = 0.0277

In [105]: m = 400

In [106]: ϕ = 0.0005

In [107]: u = 0.2

In [108]: k = k_prime / np.sqrt(m * ϕ)

In [109]: limits = [1e5, 5e5, 1e6, 2e6, 3e6, 4e6, 5e6]

In [110]: bl = build('agg Bahn.6.5 '
   .....:             f'{m * ϕ} claims '
   .....:             f'{limits} xs 0 '
   .....:             'sev exp(7) * lognorm 2.4 '
   .....:             'poisson'
   .....:             , bs=50, log2=18)
   .....:

In [111]: qd(bl.report_df.iloc[:, :-4], accuracy=4)

view               0         1         2         3         4         5         6
statistic
name        Bahn.6.5  Bahn.6.5  Bahn.6.5  Bahn.6.5  Bahn.6.5  Bahn.6.5  Bahn.6.5
limit          1e+05     5e+05     1e+06     2e+06     3e+06     4e+06     5e+06
attachment         0         0         0         0         0         0         0
el            1779.2    2725.1      3069    3347.6      3478    3556.5    3609.6
freq_m           0.2       0.2       0.2       0.2       0.2       0.2       0.2
freq_cv       2.2361    2.2361    2.2361    2.2361    2.2361    2.2361    2.2361
freq_skew     2.2361    2.2361    2.2361    2.2361    2.2361    2.2361    2.2361
sev_m           8896     13626     15345     16738     17390     17782     18048
sev_cv        2.3401    3.7719      4.63    5.6565    6.3357     6.851    7.2685
sev_skew      3.3416    7.0585    9.9524    14.303    17.849    20.975    23.827
agg_m         1779.2    2725.1      3069    3347.6      3478    3556.5    3609.6
agg_cv        5.6903    8.7257    10.592    12.844    14.342    15.482    16.406
agg_skew      8.1747    15.898    22.157    31.683    39.494    46.396    52.704
```

Next, extract the required columns from `report_df` and manipulate to compute the ILFs.

```
In [112]: bit = bl.report_df.loc[['sev_m', 'sev_cv', 'agg_m', 'agg_cv']].iloc[:, :-
↪4].T

In [113]: bit.index = limits

In [114]: bit.index.name = 'limit'

In [115]: bit['vx'] = (bit.sev_m * bit.sev_cv) ** 2

In [116]: bit['ex2'] = bit.vx + bit.sev_m**2

In [117]: bit['risk load'] = k_prime * bit.ex2 ** 0.5
```

(continues on next page)

(continued from previous page)

```
In [118]: bit['lev'] = (1+u) * bit.sev_m

In [119]: bit['ILF w/o risk'] = bit['lev'] / bit.loc[100000, 'lev']

In [120]: bit['ILF with risk'] = (bit['lev'] + bit['risk load']) / (bit.loc[100000,
↪ 'lev'] + bit.loc[100000, 'risk load'])

In [121]: qd(bit, accuracy=4)

statistic sev_m sev_cv  agg_m agg_cv         vx        ex2 risk load   lev ILF w/o␣
↪risk  \
limit                                                                              ␣
↪
100000.0   8896 2.3401 1779.2 5.6903 4.3337e+08 5.1251e+08    627.09 10675          ␣
↪   1
500000.0  13626 3.7719 2725.1 8.7257 2.6414e+09 2.8271e+09    1472.8 16351        1.
↪5316
1000000.0 15345   4.63   3069 10.592 5.0478e+09 5.2833e+09    2013.4 18414        1.
↪7249
2000000.0 16738 5.6565 3347.6 12.844  8.964e+09 9.2441e+09    2663.3 20085        1.
↪8815
3000000.0 17390 6.3357   3478 14.342 1.2139e+10 1.2442e+10    3089.7 20868        1.
↪9548
4000000.0 17782  6.851 3556.5 15.482 1.4842e+10 1.5158e+10    3410.4 21339        1.
↪9989
5000000.0 18048 7.2685 3609.6 16.406 1.7208e+10 1.7534e+10    3667.9 21658        2.
↪0288

statistic ILF with risk
limit
100000.0              1
500000.0          1.577
1000000.0        1.8074
2000000.0        2.0127
3000000.0        2.1197
4000000.0        2.1897
5000000.0        2.2407
```

Aggregate Premiums, Example 6.6

(Continues *Example 6.3*.) Compute expected losses across a variety of occurrence and aggregate limit combinations. Assume 20% ALAE outside the limits, expected claim count 1.2 with contagion parameter 0.1 (cv of mixing $\sqrt{0.1}$), and lognormal severity $(\mu, \sigma) = (7.6, 2.4)$ (see errata).

The following code calculates Table 6.4 using FFT aggregate distributions. The last column, showing unlimited aggregate losses, agrees, but the other columns are slightly different because Bahnemann uses a shifted gamma approximation.

First, we compute all the aggregates.

```
In [122]: b = {}

In [123]: for per_claim in [0.5e6, 1e6, 2e6, 3e6, 4e6, 5e6]:
   .....:     tower = np.array([0]  + [i for i in [0, 1e6, 2e6, 3e6, 4e6, 5e6, np.
↪inf]
   .....:             if i >= per_claim])
   .....:     b[per_claim] = build('agg Bahn.6.6 1.2 claims '
   .....:            f'{per_claim} xs 0 '
   .....:            'sev exp(7.6) * lognorm 2.4 '
   .....:            f'mixed gamma {0.1}**.5 '
```

(continues on next page)

(continued from previous page)

```
   .....:             f'aggregate ceded to tower {tower} '
   .....:             , bs=50, log2=18,
   .....:             normalize=False,
   .....:            )
   .....:

In [124]: qd(pd.concat([i.describe[['E[X]', 'CV(X)', 'Skew(X)']] for i in b.
↪values()],
   .....:     keys=b.keys(), names=['Occ limit', 'X']),
   .....:     accuracy=4)
   .....:

                E[X]   CV(X)   Skew(X)
Occ limit X
500000.0  Freq   1.2 0.96609   1.0696
          Sev  21743  3.1549   5.3107
          Agg  26091  3.0377   4.9943
1000000.0 Freq   1.2 0.96609   1.0696
          Sev  25279  3.8778   7.3846
          Agg  30335  3.6694   6.7822
2000000.0 Freq   1.2 0.96609   1.0696
          Sev  28338  4.7579   10.437
          Agg  34006  4.4495   9.4728
3000000.0 Freq   1.2 0.96609   1.0696
          Sev  29851  5.3511   12.892
          Agg  35821  4.9795    11.66
4000000.0 Freq   1.2 0.96609   1.0696
          Sev  30791  5.8075    15.04
          Agg  36949  5.3888   13.582
5000000.0 Freq   1.2 0.96609   1.0696
          Sev  31444  6.1815    16.99
          Agg  37733   5.725   15.332
```

Next, manipulate the output to determine layer loss costs using the `reinsurance_audit_df` dataframe. It tracks statistics for gross, ceded, and net loss across all requested layers, separately for occurrence and aggregate. In this case there are no occurrence layers. This step takes longer than computing the aggregates!

```
In [125]: bit = pd.concat([i.reinsurance_audit_df['ceded'].iloc[:-1]
   .....:                  for i in b.values()], keys=b.keys(),
   .....:                  names=['Occ limit', 'kind', 'share', 'limit', 'attach'])
   .....:

In [126]: bit['Agg limit'] = bit.index.get_level_values('limit') + bit.index.get_
↪level_values('attach')

In [127]: bit = bit.droplevel(['kind', 'share', 'limit', 'attach'])

In [128]: bit = bit.set_index('Agg limit', append=True)

In [129]: bit = bit.groupby(level='Occ limit')[['ex']].cumsum()

In [130]: el = bit.unstack('Agg limit').droplevel(0, axis=1)

In [131]: table = pd.concat((el, el / el.loc[500000, np.inf]),
   .....:                    keys=['Loss', 'ILF'])
   .....:

In [132]: qd(table.fillna(' - '), accuracy=4)

Agg limit      1000000.0 2000000.0 3000000.0 4000000.0  5000000.0   inf
     Occ limit
```

(continues on next page)

(continued from previous page)

```
Loss 500000.0      26088     26091     26091     26091      26091      26091
     1000000.0     30133     30334     30335     30335      30335      30335
     2000000.0         -     33910     33998     34006      34006      34006
     3000000.0         -         -     35761     35813      35819      35820
     4000000.0         -         -         -     36908      36942      36949
     5000000.0         -         -         -         -      37703      37733
ILF  500000.0    0.99988         1         1         1          1          1
     1000000.0    1.1549    1.1626    1.1627    1.1627     1.1627     1.1627
     2000000.0         -    1.2997    1.3031    1.3034     1.3034     1.3034
     3000000.0         -         -    1.3707    1.3726     1.3728     1.3729
     4000000.0         -         -         -    1.4146     1.4159     1.4162
     5000000.0         -         -         -         -     1.4451     1.4462
```

Here is a reconciliation to Table 6.4 of the 2M per claim and 2M aggregate limit expected loss, using the shifted gamma approximation. The limited aggregate loss is computed using the integral of the survival function `fz.sf`. `quad` is a general purpose numerical integration routine. It returns the integral and estimated error.

```
In [133]: fz = b[2000000].approximate('sgamma')

In [134]: print(fz.stats())
(34005.90148815674, 22895161300.37269)

In [135]: mv(b[2000000])
mean     = 34006.1
variance = 2.289516e+10
std dev  = 151311

In [136]: from scipy.integrate import quad

In [137]: quad(fz.sf, 0, 2000000)
Out[137]: (33523.06658768903, 0.000447320519015193)
```

Deductible Credits, Example 6.7

(Continues *Example 6.3*.) Consider a portfolio of policies for which the ground-up indemnity claim size has a lognormal distribution with parameters $(\mu, \sigma) = (7.0, 2.4)$ and allocated loss adjustment expense is 20% of the indemnity amount. The basic limit is 100,000. Calculate the credit factors, as well as the resulting frequency and severity, for six straight deductible options: 1,000; 2,000; 3,000; 4,000; 5,000; and 10,000. Base frequency equals 0.0005.

We can build all of the required distributions simultaneously using vectorization. Remember that the basic limit is ground up. The severity is unconditional, indicated by `!` at the end of the severity clause. The limit is eroded by the deductible.

```
In [138]: deductibles = [0, 1e3, 2e3, 3e3, 4e3, 5e3, 10e3]

In [139]: limits = [100000 - i for i in deductibles]

In [140]: ϕ = 0.0005

In [141]: alae = 1.2

In [142]: bl = build('agg Bahn.6.7 '
   .....:            f'{ϕ} claims '
   .....:            f'{limits} xs {deductibles} '
   .....:            'sev exp(7) * lognorm 2.4 ! '
   .....:            'poisson'
   .....:            , bs=50, log2=18)
   .....:
```

(continues on next page)

(continued from previous page)

```
In [143]: qd(bl.report_df.iloc[:, :-4], accuracy=4)

view               0         1         2         3         4         5         6
statistic
name         Bahn.6.7  Bahn.6.7  Bahn.6.7  Bahn.6.7  Bahn.6.7  Bahn.6.7  Bahn.6.7
limit           1e+05     99000     98000     97000     96000     95000     90000
attachment          0      1000      2000      3000      4000      5000     10000
el              4.448    4.1183    3.8926    3.7092    3.5517    3.4124    2.8758
freq_m         0.0005    0.0005    0.0005    0.0005    0.0005    0.0005    0.0005
freq_cv        44.721    44.721    44.721    44.721    44.721    44.721    44.721
freq_skew      44.721    44.721    44.721    44.721    44.721    44.721    44.721
sev_m            8896    8236.6    7785.2    7418.4    7103.4    6824.9    5751.7
sev_cv         2.3401    2.5106    2.6287    2.7269     2.813    2.8907    3.2067
sev_skew       3.3416    3.3681    3.4054    3.4444    3.4833    3.5215    3.6997
agg_m           4.448    4.1183    3.8926    3.7092    3.5517    3.4124    2.8758
agg_cv         113.81    120.85    125.78    129.89    133.51    136.79    150.22
agg_skew       163.49    165.89    168.03    170.02    171.89    173.66    181.54
```

Next, manipulate the `report_df` dataframe to compute the required quantities. The final exhibit replicates Table 6.5.

```
In [144]: bit = bl.report_df.iloc[:, :-4].loc[['attachment', 'freq_m', 'sev_m',
↪'agg_m']].T

In [145]: bit = bit.rename(columns={'attachment': 'deductible'}).set_index(
↪'deductible')

In [146]: bit['F(d)'] = np.array([bl.sevs[0].fz.cdf(i) for i in bit.index])

In [147]: bit['freq_m'] = bit.loc[0, 'freq_m'] * (1 - bit['F(d)'])

In [148]: bit['E[X;d]'] = (bit.sev_m[0] - bit.sev_m)

In [149]: bit['C(d)'] = bit['E[X;d]'] / bit.sev_m[0]

In [150]: bit['sev_m'] = bit['sev_m'] / (1 - bit['F(d)']) * alae

In [151]: bit = bit.iloc[:, [-2, 3, -1, 0, 1]]

In [152]: bit['pure prem'] = bit.freq_m * bit.sev_m

In [153]: qd(bit, accuracy=4)

statistic   E[X;d]    F(d)     C(d)      freq_m sev_m pure prem
deductible
0.0              0       0        0      0.0005 10675    5.3376
1000.0      659.42 0.48467 0.074125 0.00025766 19180     4.942
2000.0      1110.8 0.59885  0.12487 0.00020057 23289    4.6711
3000.0      1477.7 0.66251  0.16611 0.00016875 26377     4.451
4000.0      1792.7 0.70512  0.20151 0.00014744 28907     4.262
5000.0      2071.2 0.73636  0.23282 0.00013182 31064    4.0949
10000.0     3144.4 0.82147  0.35346 8.9266e-05 38660     3.451
```

Summary

Here is a summary of all the objects created in this section.

```
In [154]: from aggregate import pprint_ex

In [155]: for n, r in build.qlist('^Bahn').iterrows():
   .....:     pprint_ex(r.program, split=20)
   .....:
```

2.13.7 Enterprise Risk Analysis

This section re-analyzes reinsurance structure alternatives introduced in Brehm *et al.* [2007], Enterprise Risk Analysis for Property & Liability Insurance Companies. This book is the ERM text on the syllabus for CAS Exam Part 7.

Reinsurance Example

This section analyzes the example given in Section 2.5 of Enterprise Risk Analysis.

Assumptions. ABCD writes 33M excess property and casualty business.

- ABCD total gross
  - Loss ratio: 69.36%
  - Expense ratio: 23%
  - Combined ratio: 92.36%
  - Margin 2.52M
- Casualty
  - 14M premium
  - 78% expected loss ratio
  - 5M limits
  - 4M xs 1M reinsurance, ceded premium 4.41M
- Property
  - 19M premium
  - 63% expected loss ratio
  - 20M limits
  - 17M xs 3M per risk reinsurance, ceded premium 2.36M
  - 95% share of 24M xs 1M cat reinsurance, ceded premium 1.53M with 1@100%
  - Cat program designed to cover to 250-year event.
- Reinsurance total
  - Average recoveries 5.08M
  - Ceded premium 8.3M
  - Net premium 24.7M
- Mythical alternative program
  - Stop-loss, 20 xs 30, ceded premium 1.98

Additional assumptions. There are no details of the stochastic model, so we assume

- Frequency and severity models per DecL below,
- Split the property losses into cat and non-cat by assuming that cat premium equals 2M, non-cat premium 17M, at the same loss ratios (this is just a split of losses, the by line loss ratios are not used), and
- Cat, non-cat and casualty are independent.
- Free and unlimited reinstatements on the catastrophe protection. See REF for a discussion of reinstatements.

Stochastic Models and Baseline Analysis

Construct the gross and net portfolios. All amounts in millions.

Gross Portfolio

The 250-year cat PML is printed last, to compare with the 25M program.

```
In [1]: from aggregate import build, qd, mv

In [2]: import pandas as pd

In [3]: import matplotlib.pyplot as plt

In [4]: abcd = build('port ABCD '
   ...:           'agg Casualty 14.0 premium at 78% lr '
   ...:               '5 xs 0 '
   ...:               'sev lognorm 0.1 cv 10 '
   ...:               'mixed gamma 0.3 '
   ...:           'agg PropertyNC 17.0 premium at 63% lr '
   ...:               '25 xs 0 '
   ...:               'sev lognorm [0.1 1] cv [5 10] wts [.7 .3] '
   ...:               'mixed gamma 0.1 '
   ...:           'agg PropertyC 2.0 premium at 63% lr '
   ...:               '150 xs 0 '
   ...:               'sev 3 * pareto 2.375 - 3 '
   ...:               'poisson ', bs=1/128, approximation='exact')
   ...:

In [5]: qd(abcd)

                    E[X]  Est E[X]     Err E[X]   CV(X)  Est CV(X)   Skew(X)  Est␣
↪Skew(X)
unit       X                                                                     ␣
↪
Casualty   Freq   125.93                       0.31296              0.60054      ␣
↪
           Sev  0.086717 0.086436  -0.0032438  4.0147     4.0286    9.5592      9.
↪5552
           Agg     10.92    10.885  -0.0032438 0.47532   0.47626  0.88481     0.
↪88565
PropertyNC Freq   34.918                       0.19657              0.24744      ␣
↪
           Sev   0.30672   0.30659 -0.00041535    4.91     4.9121    11.569      11.
↪569
           Agg     10.71    10.706 -0.00041535 0.85384   0.85419   1.9358      1.
↪9358
PropertyC  Freq   0.5801                        1.3129               1.3129      ␣
↪
           Sev     2.172     2.172 -9.2692e-07  2.0207     2.0207   11.511      11.
↪511
           Agg      1.26      1.26 -9.2692e-07  2.9601     2.9601   12.398      12.
```

(continues on next page)

(continued from previous page)

```
↪398
total      Freq   161.42                      0.24785            0.57516           ␣
↪
           Sev    0.1418  0.14155  -0.0017419  5.8043             23.386           ␣
↪
           Agg     22.89    22.85  -0.0017419 0.48741   0.48813   1.6182      1.
↪6193
log2 = 16, bandwidth = 1/128, validation: fails sev mean, agg mean.

In [6]: mv(abcd)
mean     = 22.89
variance = 124.4768
std dev  = 11.1569

In [7]: print(abcd['PropertyC'].q(0.996))
22.859375
```

Net Portfolio

```
In [8]: abcd_net = build('port ABCD:Net '
   ...:          'agg Casualty 14.0 premium at 78% lr '
   ...:              '5 xs 0 '
   ...:              'sev lognorm 0.1 cv 10 '
   ...:              'occurrence net of 4 xs 1 '
   ...:              'mixed gamma 0.3 '
   ...:          'agg PropertyNC 17.0 premium at 63% lr '
   ...:              '25 xs 0 '
   ...:              'sev lognorm [0.1 1] cv [5 10] wts [.7 .3] '
   ...:              'occurrence net of 17 xs 3 '
   ...:              'mixed gamma 0.1 '
   ...:          'agg PropertyC 2.0 premium at 63% lr '
   ...:              '150 xs 0 '
   ...:              'sev 3 * pareto 2.375 - 3 '
   ...:              'occurrence net of 24 xs 1 '
   ...:              'poisson ', bs=1/128, approximation='exact')
   ...:

In [9]: qd(abcd_net)

                    E[X] Est E[X] Err E[X]   CV(X) Est CV(X)   Skew(X) Est Skew(X)
unit       X
Casualty   Freq   125.93                   0.31296            0.60054
           Sev  0.086717 0.065564 -0.24393  4.0147    2.5318   9.5592      4.1714
           Agg     10.92   8.2563 -0.24393 0.47532    0.3858  0.88481     0.65534
PropertyNC Freq   34.918                   0.19657            0.24744
           Sev   0.30672   0.2098 -0.31597    4.91    2.8604   11.569      6.1847
           Agg     10.71   7.3259 -0.31597 0.85384   0.52244   1.9358      1.0361
PropertyC  Freq   0.5801                    1.3129             1.3129
           Sev     2.172  0.80417 -0.62976  2.0207    2.7345   11.511      36.228
           Agg      1.26   0.4665 -0.62976  2.9601    3.8228   12.398      40.649
total      Freq   161.42                   0.24785            0.57516
           Sev    0.1418 0.099419 -0.29888  5.8043             23.386
           Agg     22.89   16.049 -0.29888 0.48741   0.32957   1.6182      2.0938
log2 = 16, bandwidth = 1/128, validation: n/a, reinsurance.

In [10]: qd(abcd_net.est_sd)
5.2892
```

Ceded Portfolio

```
In [11]: abcd_ceded = build('port ABCD:Ceded '
   ....:          'agg Casualty 14.0 premium at 78% lr '
   ....:              '5 xs 0 '
   ....:              'sev lognorm 0.1 cv 10 '
   ....:              'occurrence ceded to 4 xs 1 '
   ....:              'mixed gamma 0.3 '
   ....:          'agg PropertyNC 17.0 premium at 63% lr '
   ....:              '25 xs 0 '
   ....:              'sev lognorm [0.1 1] cv [5 10] wts [.7 .3] '
   ....:              'occurrence ceded to 17 xs 3 '
   ....:              'mixed gamma 0.1 '
   ....:          'agg PropertyC 2.0 premium at 63% lr '
   ....:              '150 xs 0 '
   ....:              'sev 3 * pareto 2.375 - 3 '
   ....:              'occurrence ceded to 24 xs 1 '
   ....:              'poisson ', bs=1/128, approximation='exact')
   ....:

In [12]: qd(abcd_ceded)

                    E[X] Est E[X] Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit        X
Casualty    Freq   125.93                  0.31296            0.60054
            Sev  0.086717 0.020872 -0.75931  4.0147    11.205   9.5592      14.019
            Agg     10.92   2.6283 -0.75931 0.47532    1.0464  0.88481      1.3572
PropertyNC  Freq   34.918                  0.19657            0.24744
            Sev   0.30672 0.096787 -0.68444    4.91    10.657   11.569       13.51
            Agg     10.71   3.3796 -0.68444 0.85384    1.8142   1.9358      2.3095
PropertyC   Freq   0.5801                   1.3129             1.3129
            Sev     2.172   1.3679 -0.37024  2.0207     2.254   11.511      4.2512
            Agg      1.26   0.7935 -0.37024  2.9601    3.2375   12.398      5.6851
total       Freq   161.42                  0.24785            0.57516
            Sev    0.1418 0.042134 -0.70286  5.8043             23.386
            Agg     22.89   6.8014 -0.70286 0.48741    1.0577   1.6182      1.7643
log2 = 16, bandwidth = 1/128, validation: n/a, reinsurance.

In [13]: qd(abcd_ceded.est_sd)
7.1941
```

Reinsurance Summary

The bottom table shows expected losses, counts, severity, loss ratios and margins implicit in the given reinsurance structure, pricing, and the gross stochastic model. The non-cat property reinsurance has the highest ceded loss ratio and the cat program the lowest.

```
In [14]: re_all = pd.concat((a.reinsurance_occ_layer_df for a in abcd_net),
   ....:     keys=abcd_net.unit_names, names=['unit', 'share', 'limit', 'attach']);
↪ \
   ....: re_all = re_all.drop('gup', axis=0, level=3); \
   ....: qd(re_all, sparsify=False)
   ....:

stat                             ex      ex       ex      cv      cv       cv       en␣
↪severity     pct
view                          ceded     net  subject   ceded     net  subject    ceded ␣
↪  ceded    ceded
unit        share limit attach                                                          ␣
↪
```

(continues on next page)

(continued from previous page)

```
Casualty    1.0    4.0    1.0     2.6283 8.2563  10.885 11.205 2.5318   4.0286    2.007 ␣
↪ 1.3096 0.24147
PropertyNC 1.0    17.0   3.0     3.3796 7.3259  10.706 10.657 2.8604   4.9121 0.65611 ␣
↪  5.151 0.31569
PropertyC  1.0    24.0   1.0     0.7935 0.4665    1.26  2.254 2.7345   2.0207 0.29294 ␣
↪ 2.7088 0.62976

In [15]: re_summary = re_all.iloc[:, [0, 3, 6, 7]]; \
   ....: re_summary.columns = ['ex', 'cv', 'en', 'severity']; \
   ....: re_summary['premium'] = [4.41, 2.36, 1.53]; \
   ....: re_summary['lr'] = re_summary.ex / re_summary.premium; \
   ....: re_summary['margin'] = re_summary.premium - re_summary.ex; \
   ....: qd(re_summary)
   ....:

                                  ex      cv       en  severity  premium       lr ␣
↪margin
unit       share limit  attach                                                        ␣
↪
Casualty    1.0    4.0    1.0     2.6283 11.205   2.007    1.3096      4.41 0.59599  1.
↪7817
PropertyNC 1.0    17.0   3.0     3.3796 10.657 0.65611     5.151      2.36   1.432 -1.
↪0196
PropertyC  1.0    24.0   1.0     0.7935  2.254 0.29294    2.7088      1.53 0.51863  0.
↪7365
```

Underwriting Result Distributions

Make the underwriting result distributions, including the proposed stop loss reinsurance (computed by hand). The dataframe `compare` accumulates the gross, ceded, and net probability mass functions. We use these to determine statistics and to plot.

```
In [16]: compare = abcd.density_df[['loss', 'p_total']]; \
   ....: compare.columns = ['loss', 'gross']; \
   ....: compare['gross_uw'] = 33 - compare.loss; \
   ....: compare['net_current'] = abcd_net.density_df.p_total; \
   ....: compare['net_current_uw'] = 33 - 4.41 - 2.36 - 1.53 - compare.loss;
   ....:

In [17]: from aggregate import make_ceder_netter

In [18]: compare['net_stoploss'] = abcd.density_df.p_total; \
   ....: c, n = make_ceder_netter([(1, 20, 30)]); \
   ....: compare['nsll'] = n(compare.loss); \
   ....: g = compare.groupby('nsll').net_stoploss.sum(); \
   ....: compare['net_stoploss'] = 0.0; \
   ....: compare.loc[g.index, 'net_stoploss'] = g; \
   ....: compare['net_stoploss_uw'] = 33 - 1.98 - compare.loss;
   ....:
```

Comparison with ERA Book Figures

Statistics summary, compare Figure 2.5.2.

```
In [19]: from aggregate import MomentWrangler

In [20]: from scipy.interpolate import interp1d

In [21]: ans = []; cdfs = []

In [22]: for xs, den in [(compare.gross_uw, compare.gross), (compare.net_current_
↪uw, compare.net_current),
   ....:                    (compare.net_stoploss_uw, compare.net_stoploss)]:
   ....:     xd = xs * den
   ....:     ex1 = np.sum(xd)
   ....:     xd *= xs
   ....:     ex2 = np.sum(xd)
   ....:     ex3 = np.sum(xd * xs)
   ....:     mw = MomentWrangler()
   ....:     mw.noncentral = ex1, ex2, ex3
   ....:     ans.append(mw)
   ....:     cdfs.append(interp1d(den.cumsum(), xs))
   ....:

In [23]: fig_252 = pd.concat([i.stats for i in ans], keys=['Gross', 'Current',
↪'StopLoss'], axis=1)

In [24]: for p in [0.01, 0.99]:
   ....:     fig_252.loc[f'q({p})'] = [float(i(p)) for i in cdfs]
   ....:

In [25]: qd(fig_252)

           Gross  Current  StopLoss
ex         10.15   8.6513    10.021
var       124.41   27.975    58.777
sd        11.154   5.2892    7.6666
cv        1.0989  0.61137   0.76502
skew     -1.6193  -2.0938   -1.0088
q(0.01)    25.98   18.081        24
q(0.99)   -25.76  -5.6036   -7.7399
```

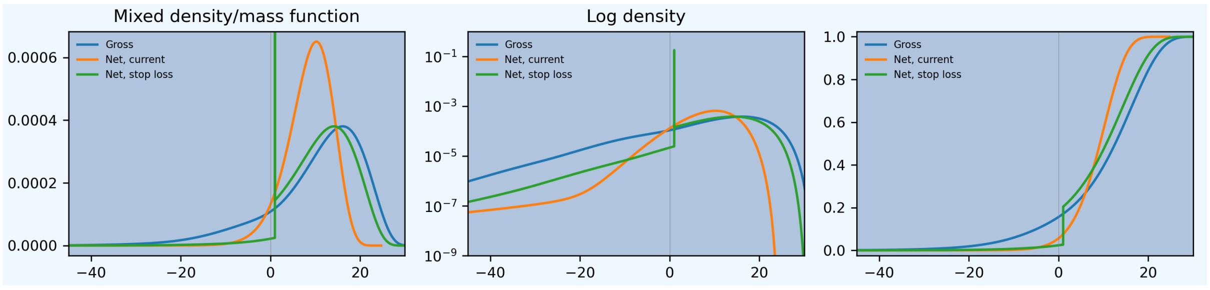
Plot of densities and distributions, compare Figure 2.5.3 and 2.5.4.

```
In [26]: fig, axs = plt.subplots(1, 3, figsize=(3 * 3.5, 2.45), constrained_
↪layout=True)

In [27]: ax0, ax1, ax2 = axs.flat

In [28]: for ax in [ax0, ax1]:
   ....:     ax.plot(compare.gross_uw, compare.gross, label='Gross')
   ....:     ax.plot(compare.net_current_uw, compare.net_current, label='Net,␣
↪current')
   ....:     yl = ax.get_ylim()
   ....:     ax.plot(compare.net_stoploss_uw, compare.net_stoploss, label='Net,␣
↪stop loss')
   ....:     ax.legend(loc='upper left')
   ....:     ax.set(xlim=[-45, 30], ylim=yl)
   ....:     ax.axvline(0, lw=.25, c='C7')
   ....:

In [29]: ax1.set(yscale='log', ylim=[1e-9, 1], title='Log density'); \
```

(continues on next page)

(continued from previous page)

```
   ....: ax0.set(title='Mixed density/mass function');
   ....:

In [30]: ax2.plot(compare.gross_uw, 1 - compare.gross.cumsum(), label='Gross'); \
   ....: ax2.plot(compare.net_current_uw, 1 - compare.net_current.cumsum(), label=
↪'Net, current'); \
   ....: ax2.plot(compare.net_stoploss_uw, 1 - compare.net_stoploss.cumsum(),␣
↪label='Net, stop loss'); \
   ....: ax2.legend(loc='upper left'); \
   ....: ax2.set(xlim=[-45, 30], ylim=[-0.025, 1.025]);
   ....:

In [31]: ax2.axvline(0, lw=.25, c='C7');
```



Numerical distribution of underwriting results at various return points, compare Figure 2.5.5. Given there was no information about the stochastic model provided, and the model here is based on common benchmarks, the agreement between the two distributions is striking.

```
In [32]: fig_255 = pd.DataFrame(columns=['Gross', 'Current', 'StopLoss'],␣
↪dtype=float)

In [33]: for p in [.0025, .005, 0.0075, .01, .0125, .015, .0175, .02,
   ....:             .04, .06, .08, .1, .12, .14, .16, .18, .2, .22, .24,
   ....:             .25, .26, .28, .3, .32, .34, .36, .38, .4, .42, .44,
   ....:             .46, .48, .5]:
   ....:     fig_255.loc[p] = [float(i(1-p)) for i in cdfs]
   ....:

In [34]: fig_255.index.name = 'p'

In [35]: qd(fig_255, float_format=lambda x: f'{x:.3f}', max_rows=len(fig_255))

         Gross  Current  StopLoss
p
0.0025 -38.503  -10.257   -20.483
0.0050 -32.113   -7.826   -14.093
0.0075 -28.394   -6.514   -10.374
0.0100 -25.760   -5.604    -7.740
0.0125 -23.740   -4.902    -5.720
0.0150 -22.114   -4.328    -4.094
0.0175 -20.758   -3.842    -2.738
0.0200 -19.595   -3.420    -1.575
0.0400 -13.603   -1.182     1.021
0.0600  -9.962    0.182     1.021
0.0800  -7.211    1.186     1.022
0.1000  -4.936    1.991     1.023
0.1200  -2.978    2.669     1.024
0.1400  -1.263    3.259     1.025
0.1600   0.246    3.784     1.026
0.1800   1.575    4.259     1.027
0.2000   2.752    4.695     1.028
```

(continues on next page)

(continued from previous page)

```
0.2200   3.801    5.100     1.821
0.2400   4.745    5.478     2.765
0.2500   5.183    5.659     3.203
0.2600   5.603    5.835     3.623
0.2800   6.391    6.173     4.411
0.3000   7.121    6.496     5.141
0.3200   7.802    6.805     5.822
0.3400   8.442    7.103     6.462
0.3600   9.047    7.390     7.067
0.3800   9.622    7.669     7.642
0.4000  10.171    7.941     8.191
0.4200  10.697    8.206     8.717
0.4400  11.203    8.465     9.223
0.4600  11.692    8.720     9.712
0.4800  12.167    8.970    10.187
0.5000  12.628    9.217    10.648
```

Modern Analysis

The first step is to analyze the pricing in the context of needed capital. Strip expenses out (at 23% across all units) to determine a net (of expenses) technical premium.

```
In [36]: er = 0.23

In [37]: df = pd.DataFrame({'unit': ['Casualty', 'PropertyNC', 'PropertyC'],
   ....:                                'prem': [14, 17, 2],
   ....:                                  'gross_loss': [a.est_m for a in abcd]}).set_
↪index('unit')
   ....:

In [38]: df['ceded_prem'] = [4.41, 2.36, 1.53]; \
   ....: df['net_prem'] = df.prem - df.ceded_prem; \
   ....: df['tech_prem'] = df.prem * (1 - er); \
   ....: df['margin'] = df.tech_prem - df.gross_loss; \
   ....: df.loc['Total'] = df.sum(0); \
   ....: df['lr'] = df.gross_loss / df.prem; \
   ....: df['cr'] = df.lr + er; \
   ....: df['tech_lr'] = df.gross_loss / df.tech_prem;
   ....:

In [39]: fp = lambda x: f'{x:.1%}';

In [40]: fc = lambda x: f'{x:.2f}'

In [41]: qd(df, float_format=fc, formatters={'lr':fp, 'cr': fp, 'tech_lr': fp})

             prem  gross_loss  ceded_prem  net_prem  tech_prem  margin    lr      cr␣
↪tech_lr
unit                                                                                ␣
↪
Casualty    14.00       10.88        4.41      9.59      10.78   -0.10 77.7% 100.7%␣
↪ 101.0%
PropertyNC 17.00       10.71        2.36     14.64      13.09    2.38 63.0%  86.0%␣
↪  81.8%
PropertyC   2.00        1.26        1.53      0.47       1.54    0.28 63.0%  86.0%␣
↪  81.8%
Total      33.00       22.85        8.30     24.70      25.41    2.56 69.2%  92.2%␣
↪  89.9%
```

The example does not specify a capital standard. Let’s investigate the implied return on capital at different capital standards. The capital standard is expressed as a loss percentile. The next calculation produces a table of returns expressed as a cost of capital (`coc`). It also shows the expected policyholder deficit.

```
In [42]: tech_prem = df.loc['Total', 'tech_prem']; \
   ....: ps = [.99, .995, .996, .999]; \
   ....: As = [abcd.q(p) for  p in ps]; \
   ....: el = abcd.density_df.loc[As, 'exa_total']; \
   ....: margin = tech_prem - el; \
   ....: cocs = margin /  (As - tech_prem); \
   ....: summary = pd.DataFrame({'p': ps, 'a': As, 'prem':tech_prem, 'el': el,
   ....:                          'margin': margin, 'tech_lr': el / tech_prem, 'coc
↪': cocs,
   ....:                         'epd': (abcd.est_m - el) / abcd.est_m}).set_index(
↪'p')
   ....:

In [43]: summary.index = [fp(i) for i in summary.index]; \
   ....: summary.index.name = 'p'; \
   ....: qd(summary, float_format=fc, formatters={'coc': fp, 'tech_lr': fp, 'epd':␣
↪fp})
   ....:

           a  prem    el  margin tech_lr  coc  epd
p
99.0% 58.77 25.41 22.75    2.66   89.5% 8.0% 0.5%
99.5% 65.12 25.41 22.79    2.62   89.7% 6.6% 0.3%
99.6% 67.16 25.41 22.80    2.61   89.7% 6.2% 0.2%
99.9% 80.90 25.41 22.83    2.58   89.8% 4.6% 0.1%
```

Based on this analysis, we assume a 99.5% (200-year) capital standard, which gives a reasonable 8% return on capital. 200-year capital is also the Solvency II standard.

From here, the analysis could proceed in many directions. The approach we select is

1. Calibrate a set of distortions to total pricing on a gross basis with the 200-year capital standard.
2. Analyze the pricing implied by these distortions on the net book and its natural allocation by unit.
3. Compare the model value (implied ceded premium) with market reinsurance price.

The model value is the *maximum* amount that is consistent with pricing according to each distortion. Reinsurance cheaper than the model value is efficient: replacing traditional capital with reinsurance capital lowers the economic cost of bearing risk.

Calibrate Distortions

Extract the exact cost of capital implied by given gross pricing.

```
In [44]: coc = summary.loc['99.5%', 'coc']

In [45]: print(coc)
0.06591531668710866
```

Calibrate distortions to current pricing. Use five one-parameter distortion families

1. constant cost of capital (CCoC),
2. proportional hazard (PH)
3. Wang,
4. dual, and
5. TVaR.

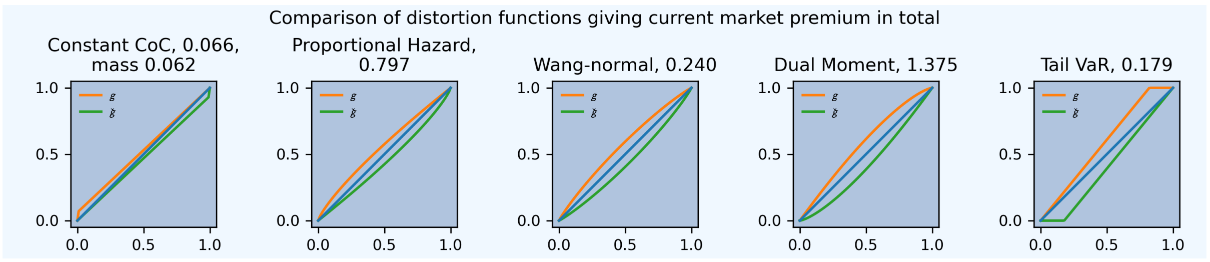
They are sorted from most tail-centric (expensive for tail risk) to cheapest. See Mildenhall and Major [2022].

The next dataframe shows the asset level and implied loss ratio, distortion name, survival probability (0.5%), expected loss, premium, premium to capital leverage (`PQ`), the cost of (return on) capital, the distortion family parameter, and the parameterization error. The calibrated premium matches the technical premium.

```
In [46]: abcd.calibrate_distortions(ROEs=[coc], Ps=[.995], strict='ordered');

In [47]: qd(abcd.distortion_df)

                                              S        L       P        PQ       Q        COC     param ␣
↪      error
a          LR       method                                                                         ␣
↪
65.117188 0.896997 ccoc   0.0049978 22.793 25.41 0.63993 39.707 0.065915 0.065915 ␣
↪          0
                   ph     0.0049978 22.793 25.41 0.63993 39.707 0.065915   0.7966 ␣
↪4.7227e-06
                   wang   0.0049978 22.793 25.41 0.63993 39.707 0.065915  0.24044 ␣
↪1.1474e-06
                   dual   0.0049978 22.793 25.41 0.63993 39.707 0.065915   1.3749 -
↪7.4227e-11
                   tvar   0.0049978 22.793 25.41 0.63993 39.707 0.065915  0.17864 ␣
↪9.4873e-06
```

The plot show this effect: COC is fattest on the left for small exceedance probabilities (high losses), whereas TVaR is fattest on the right.

```
In [48]: fig, axs = plt.subplots(1, 5, figsize=(10.0, 2.1), constrained_
↪layout=True)

In [49]: for ax, (k, v) in zip(axs.flat, abcd.dists.items()):
   ....:     v.plot(ax=ax)
   ....:

In [50]: fig.suptitle('Comparison of distortion functions giving current market␣
↪premium in total')
Out[50]: Text(0.5, 0.98, 'Comparison of distortion functions giving current market␣
↪premium in total')
```



Analyze Implied Pricing

Apply the distortions to the net portfolio and analyze the resulting pricing using `analyze_distortions()`, which includes a by-unit margin allocation. The dataframe `ans.comp_df` contains a wealth of other information; we just focus on the premium. The last row, `Technical`, shows market reinsurance pricing.

```
In [51]: abcd_net.dists = abcd.dists

In [52]: ansn = abcd_net.analyze_distortions(p=0.996, add_comps=False); \
   ....: ans = abcd.analyze_distortions(p=0.996, add_comps=False); \
   ....: bit = pd.concat((ans.comp_df.xs('P', 0, 1), ansn.comp_df.xs('P', 0, 1),
```

(continues on next page)

(continued from previous page)

```
   ....:                  ans.comp_df.xs('P', 0, 1) - ansn.comp_df.xs('P', 0, 1)),
   ....:                  axis=1, keys=['gross', 'net', 'ceded']); \
   ....: bit = bit.iloc[[0, 2,-1, 1, -2]]; \
   ....: bit.loc['Technical'] = 0.0; \
   ....: bit.loc['Technical', 'gross'] = df.tech_prem.sort_index().values; \
   ....: bit.loc['Technical', 'ceded'] = df.ceded_prem.sort_index().values; \
   ....: bit.loc['Technical', 'net'] = df.net_prem.sort_index().values; \
   ....: qd(bit, sparsify=False, line_width=50)
   ....:

             gross     gross      gross  gross  \
line      Casualty PropertyC PropertyNC  total
Method
Dist ccoc   10.371    4.6237      10.55 25.545
Dist ph     11.347    1.5483     12.542 25.438
Dist wang   11.514    1.4754     12.439 25.428
Dist dual   11.698    1.4415     12.283 25.423
Dist tvar   11.908    1.4298     12.084 25.421
Technical    10.78      1.54      13.09  25.41

               net       net        net    net  \
line      Casualty PropertyC PropertyNC  total
Method
Dist ccoc   7.8043    2.3449     6.9327 17.082
Dist ph     8.6519   0.53481      7.977 17.164
Dist wang   8.7296   0.49483     8.0137 17.238
Dist dual   8.8065   0.47931     8.0416 17.327
Dist tvar   8.8941   0.47972     8.0693 17.443
Technical     9.59      0.47      14.64   24.7

             ceded     ceded      ceded  ceded
line      Casualty PropertyC PropertyNC  total
Method
Dist ccoc   2.5669    2.2788     3.6172 8.4632
Dist ph     2.6952    1.0135     4.5651 8.2738
Dist wang    2.784   0.98056     4.4253 8.1898
Dist dual   2.8917   0.96217     4.2413 8.0952
Dist tvar   3.0137   0.95009     4.0143 7.9781
Technical     4.41      1.53       2.36    8.3
```

Compare Model Value and Market Price

Focus on the last block above, under `ceded`. The rows `Dist ...` show the model value of reinsurance according to each distortion. The row `Technical` shows the market price. The market suggests to buy when the value is greater than the price.

The analysis provides a clear answer only for casualty, where the model value of reinsurance is much lower than the market price for all distortions: don't buy the reinsurance.

For property cat, CCoC, the most tail-centric distortion, sees a lot of value in the reinsurance — hardly surprising. All the other less tail-centric distortions do not see it as adding value overall (lower value than market price). The order of the distortions and their assessment of the value of cat reinsurance are perfectly aligned, as they were for casualty albeit in the opposite order.

For property non-cat, the PH and Wang distortions see value, the others do not, though dual is close. This is the most interesting case because the ranking does not agree with the distortion ordering (as it does for the other two units). Property non-cat contributes to volatility and tail-risk, and so is more nuanced. Management often struggles with property risk reinsurance because tail-centric measures understate the value it provides. Actuaries stuggle to find analytic methods that capture its management-perceived value. The range of distortions considered covers the two views well.

In total the program is not seen as good value by any of the distortions. Since they span the reasonable range of risk preferences, this is a robust result.

Management often cares about more than just tail risk and they generally rejects the findings from CCoC. Whether or not they see value in reinsurance is sensitive to their exact risk appetite. These findings are consistent with the fact that each company tends to structure its reinsurance differently, tailored to their own risk appetite. Difference in risk appetite have a material impact on decision making.

Analysis for Stop Loss Reinsurance

Here is the analysis for the stop loss reinsurance. This analysis is manual, because the net of stop loss distribution for a `Portfolio` is not currently built-in. We have to extract the relevant distributions and apply the distortions, estimate `a_stoploss` the net asset requirement at `p=0.995` (rounded to be a multiple of `bs`), determine the net expected loss and the model value. Recall `compare.net_stoploss` is the density of the net of stop-loss loss outcome. `S1` is used to create its survival function, to which the distortion is applied to determine pricing. `exa` and `exag` are the objective and risk adjusted losses (model value) given an asset level `a`, computed as $\int_0^a S$ and $\int_0^a g(S)$ respectively (see PIR REF). We then select the relevant row and assemble the answer.

```
In [53]: S0 = pd.Series(compare.net_stoploss, index=compare.loss); \
   ....: S0.name = 'S'; \
   ....: S1 = S0[::-1].shift(1, fill_value=0).cumsum(); \
   ....: a0 = float(interp1d(S0.cumsum(), S0.index)(0.995)); \
   ....: a_stoploss = abcd.snap(a0); \
   ....: print(f'Net of stoploss assets {a_stoploss:.3f}');
   ....:
Net of stoploss assets 45.109

In [54]: net_el_stoploss_unlim = (compare.loss * compare.net_stoploss).sum(); \
   ....: net_el_stoploss = (np.minimum(compare.loss, a_stoploss) * compare.net_
↪stoploss).sum(); \
   ....: epd = 1 - net_el_stoploss / net_el_stoploss_unlim; \
   ....: qd(pd.Series([net_el_stoploss_unlim, net_el_stoploss, epd], index=[
↪'unlimited net loss', 'net loss limited by assets', 'epd']));
   ....:

unlimited net loss              20.999
net loss limited by assets      20.941
epd                          0.0027373

In [55]: pricer = S1.to_frame().sort_index();

In [56]: for nm, dist in abcd.dists.items():
   ....:     pricer[f'{nm}_exa'] = pricer['S'].shift(1, fill_value=0).cumsum() *␣
↪abcd.bs
   ....:     pricer[f'{nm}_gS'] = dist.g(pricer.S)
   ....:     pricer[f'{nm}_exag'] = pricer[f'{nm}_gS'].shift(1, fill_value=0).
↪cumsum() * abcd.bs
   ....:     pricer = pricer.sort_index()
   ....:

In [57]: try:
   ....:     pricer = pricer.loc[[a_stoploss]]; \
   ....:     pricer.columns = pricer.columns.str.split('_', expand=True); \
   ....:     comp = pricer.stack(0).droplevel(0,0); \
   ....:     comp.loc['Technical'] = [net_el_stoploss, tech_prem - 1.98, np.nan]; \
   ....:     comp['stoploss_value'] = tech_prem - comp.exag; \
   ....:     comp = comp.sort_values('stoploss_value', ascending=False); \
   ....:     qd(comp)
   ....: except:
   ....:     print('Unspecfied error: TODO investigate.')
```

(continues on next page)

(continued from previous page)

```
   ....:
Unspecfied error: TODO investigate.
```

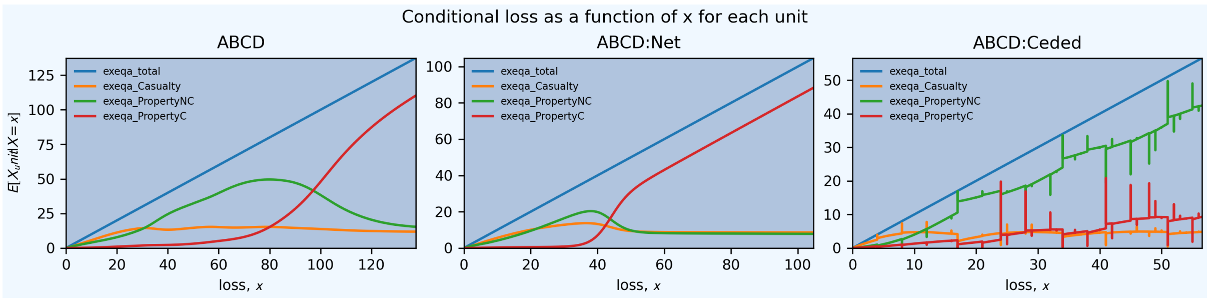
The output table reveals that the stop loss value is greater than its market price for the CCoC, PH, and Wang distortions, but less for the dual and TVaR. Thus, management averse to tail risk regard it as beneficial, but those more concerned with volatility and body risk do not see it as worthwhile.

A note of caution is in order on this analysis. Stop loss structures are a broker favorite, but are generally not liked by reinsurers. Aggregate features are hard to underwrite and price, and the lower premium is not attractive. A treaty similar to the proposed stop loss would be very hard to find in the market.

Visualizing Risk

The next figure shows the kappa functions, a handy way to visualize which units are contributing to total risk across the loss spectrum (see REF). Here the horizontal axis is total loss. The middle plot shows the reinsurance is quite effective at lowering the risk from Property NC (green line), but less effective at altering the risk profile of the other two lines. In particular, cat (red line) still dominates the tail risk.

```
In [58]: fig, axs = plt.subplots(1, 3, figsize=(3 * 3.5, 2.55), constrained_
↪layout=True)

In [59]: for ax, a in zip(axs.flat, [abcd, abcd_net, abcd_ceded]):
   ....:     mx = a.q(0.9999)
   ....:     a.density_df.filter(regex='exeqa_[CPt]').plot(ax=ax,
   ....:         xlim=[0, mx], ylim=[0, mx], title=a.name);
   ....:     ax.set(xlabel='loss, $x$');
   ....:

In [60]: axs.flat[0].set(ylabel='$E[X_unit | X=x]$');

In [61]: fig.suptitle('Conditional loss as a function of x for each unit');
```



2.13.8 Other Papers

Miscellaneous short examples from various texts.

Contents

Wang on the Wang Transform

Source paper: Wang [2000].

Pricing by Layer

Concepts: Layer expected loss and risk adjusted layer technical premium with Wang and proportional spectral risk measures.

Setup: Ground-up Pareto risk, shape 1.2, scale 2000. Layer and compare Wang(0.1) and PH(0.9245) pricing.

Source Exhibits:

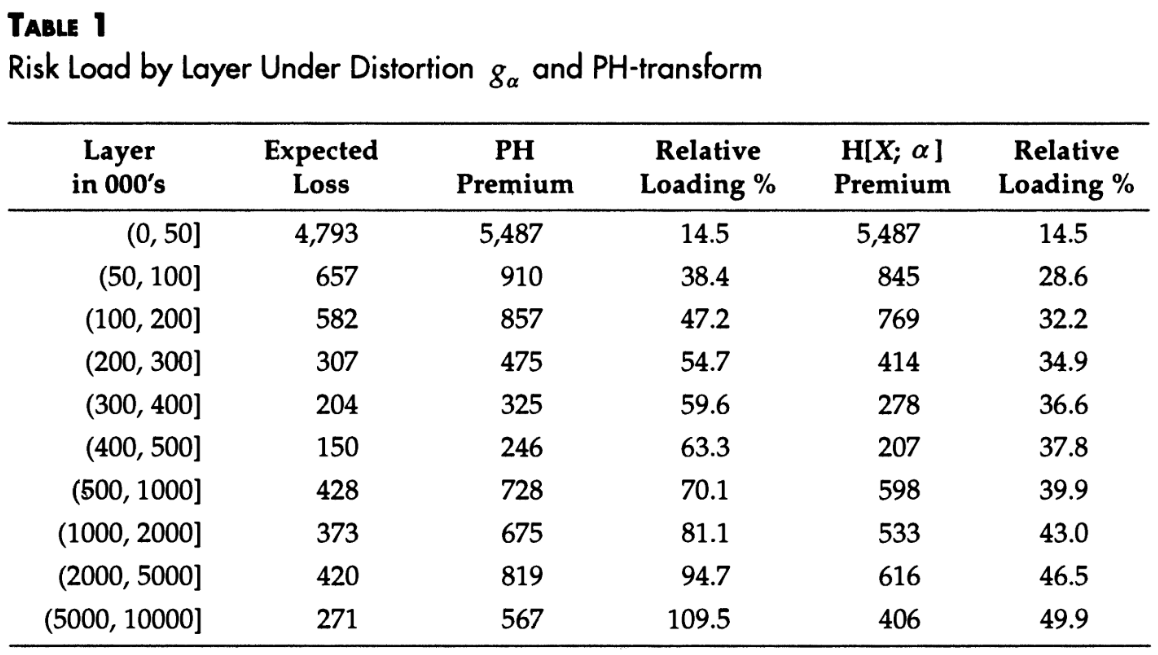


TABLE 1
Risk Load by Layer Under Distortion g_α and PH-transform

Layer in 000's	Expected Loss	PH Premium	Relative Loading %	H[X; α] Premium	Relative Loading %
(0, 50]	4,793	5,487	14.5	5,487	14.5
(50, 100]	657	910	38.4	845	28.6
(100, 200]	582	857	47.2	769	32.2
(200, 300]	307	475	54.7	414	34.9
(300, 400]	204	325	59.6	278	36.6
(400, 500]	150	246	63.3	207	37.8
(500, 1000]	428	728	70.1	598	39.9
(1000, 2000]	373	675	81.1	533	43.0
(2000, 5000]	420	819	94.7	616	46.5
(5000, 10000]	271	567	109.5	406	49.9

Thanks: Zach Eisenstein of Aon.

Code:

Build the portfolio.

```
In [1]: from aggregate import build, qd

In [2]: layers = [0, 50e3, 100e3, 200e3, 300e3, 400e3, 500e3, 1000e3, 2000e3,␣
↪5000e3, 10000e3]
```

(continues on next page)

(continued from previous page)

```
In [3]: a1 = build('agg Wang.t1 '
   ...:            '1 claim '
   ...:            '10000e3 xs 0 ' # limit the severity to avoid infinite variance
   ...:            'sev 2000 * pareto 1.2 - 2000 '
   ...:            f'occurrence ceded to tower {layers} '
   ...:            'fixed'
   ...:           )
   ...:

In [4]: qd(a1)

       E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq      1                           0
Sev  8179.5   8178.5 -0.00012201 11.595    11.596  70.841      70.841
Agg  8179.5   8178.5 -0.00012201 11.595    11.596  70.841      70.841
log2 = 16, bandwidth = 200, validation: n/a, reinsurance.
```

Expected loss and other ceded statistics by layer.

```
In [5]: ff = lambda x: x if type(x)==str else (
   ...:         f'{x/1e6:,.1f}M' if x >= 500000 else
   ...:         (f'{x:,.3f}' if x < 100 else f'{x:,.0f}'))
   ...:

In [6]: fp = lambda x: f'{x:.1%}'

In [7]: qdl = lambda x: qd(x, index=False, line_width=200, formatters={'pct': fp},
   ...:                    float_format=ff, col_space=10)
   ...:

In [8]: qdl(a1.reinsurance_occ_layer_df .xs('ceded', 1, 1).droplevel(0).reset_
↪index(drop=False))

     limit     attach         ex         cv         en   severity        pct
    50,000      0.000      4,787      1.908      1.000      4,787      58.5%
    50,000     50,000        657      8.058      0.020     32,776       8.0%
   100,000    100,000        582     12.148      0.009     65,143       7.1%
   100,000    200,000        307     17.279      0.004     78,055       3.8%
   100,000    300,000        204     21.484      0.002     83,955       2.5%
   100,000    400,000        150     25.182      0.002     87,349       1.8%
      0.5M       0.5M        428     31.751      0.001    324,046       5.2%
      1.0M       1.0M        373     48.091      0.001       0.6M       4.6%
      3.0M       2.0M        420     76.476      0.000       1.7M       5.1%
      5.0M       5.0M        271        126      0.000       3.2M       3.3%
      infM        gup      8,179     11.596      1.000      8,179     100.0%
```

ZE provided function to make the exhibit table. The column `pct` shows the relative loading.

```
In [9]: def make_table(agg, layers):
   ...:     agg_df = agg.density_df
   ...:     layer_df = agg_df.loc[layers, ['F', 'S', 'lev', 'gS', 'exag']]
   ...:     layer_df['layer_el'] = np.diff(layer_df.lev, prepend = 0)
   ...:     layer_df['premium'] = np.diff(layer_df.exag, prepend = 0)
   ...:     layer_df['pct'] = layer_df['premium'] / layer_df['layer_el'] - 1
   ...:     layer_df = layer_df.rename_axis("exhaust").reset_index()
   ...:     layer_df['attach'] = layer_df['exhaust'].shift(1).fillna(0)
   ...:     qdl(layer_df.loc[1:, ['attach', 'exhaust', 'layer_el', 'premium', 'pct
↪']])
   ...:
```

Make the distortions and apply. First, Wang.

```
In [10]: d1 = build('distortion wang_d1 wang 0.1')

In [11]: a1.apply_distortion(d1)

In [12]: make_table(a1, layers)

    attach    exhaust   layer_el    premium        pct
     0.000     50,000      4,787      5,481      14.5%
    50,000    100,000        657        845      28.6%
   100,000    200,000        582        769      32.2%
   200,000    300,000        307        414      34.9%
   300,000    400,000        204        278      36.6%
   400,000       0.5M        150        207      37.8%
      0.5M       1.0M        428        598      39.9%
      1.0M       2.0M        373        533      43.0%
      2.0M       5.0M        420        616      46.5%
      5.0M      10.0M        271        406      49.9%
```

Make the distortions and apply. Second, PH.

```
In [13]: d2 = build('distortion wang_d2 ph 0.9245')

In [14]: a1.apply_distortion(d2)

In [15]: make_table(a1, layers)

    attach    exhaust   layer_el    premium        pct
     0.000     50,000      4,787      5,480      14.5%
    50,000    100,000        657        910      38.4%
   100,000    200,000        582        856      47.2%
   200,000    300,000        307        475      54.7%
   300,000    400,000        204        325      59.6%
   400,000       0.5M        150        246      63.3%
      0.5M       1.0M        428        727      70.1%
      1.0M       2.0M        373        675      81.1%
      2.0M       5.0M        420        818      94.7%
      5.0M      10.0M        271        567     109.5%
```

It appears the layer 50000 xs 0 is reported incorrectly in the paper.

Satellite Pricing

Concepts: The cost of a Bernoulli risk with a 5% probability of a total loss of $100m using Wang(0.1) distortion.

Thanks: Zach Eisenstein of Aon.

Code:

Build the portfolio. Illustrates how to set up a Bernoulli.

```
In [16]: a2 = build('agg wang2 0.05 claims dsev [100] bernoulli')

In [17]: qd(a2)

      E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq  0.05                      4.3589             4.1295
Sev    100      100           0      0          0
Agg      5        5 8.8818e-16 4.3589     4.3589  4.1295      4.1295
log2 = 8, bandwidth = 1, validation: not unreasonable.
```

Check the distribution output.

```
In [18]: qd(a2.density_df.query('p_total > 0')[['p_total', 'F', 'S']])

       p_total    F    S
loss
0.0       0.95 0.95 0.05
100.0     0.05    1    0
```

Build and apply the distortion. Use the `price()` method. The first argument selects the 100% quantile for pricing, i.e., the limit is fully collateralized. First with the Wang from above. The relative loading reported is the complement of the loss ratio.

```
In [19]: qd(a2.price(1, d1))

statistic  L      P      M      Q   a      LR       PQ     ROE
line
wang2      5 6.1191 1.1191 93.881 100 0.81712 0.065179 0.01192
```

Try with a more severe distortion.

```
In [20]: d3 = build('distortion wang_d2 wang 0.15')

In [21]: qd(a2.price(1, d2))

statistic  L     P     M      Q   a      LR       PQ      ROE
line
wang2      5 6.269 1.269 93.731 100 0.79758 0.066883 0.013539
```

Wang on Weather Derivatives

Source paper: Wang [2002].

Concepts: Applying Wang transform to empirical distribution via an `Aggregate` object.

Source Exhibits:

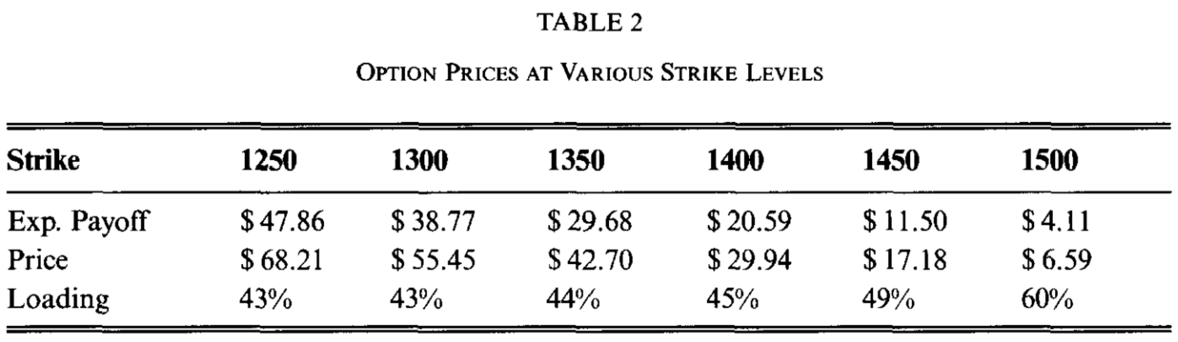


TABLE 2

OPTION PRICES AT VARIOUS STRIKE LEVELS

Strike	1250	1300	1350	1400	1450	1500
Exp. Payoff	$ 47.86	$ 38.77	$ 29.68	$ 20.59	$ 11.50	$ 4.11
Price	$ 68.21	$ 55.45	$ 42.70	$ 29.94	$ 17.18	$ 6.59
Loading	43%	43%	44%	45%	49%	60%

Data:

Create a dataframe from the heating degree days (HDD) history laid out in Table 1.

```
In [22]: d = '''Dec-79
   ....: 972.5
   ....: Dec-87
   ....: 1018.5
   ....: Dec-95
   ....: 1199.5
   ....: Dec-80
   ....: 1147.0
   ....: Dec-88
```

(continues on next page)

(continued from previous page)

```
   ....: 1155.0
   ....: Dec-96
   ....: 1156.0
   ....: Dec-81
   ....: 1244.0
   ....: Dec-89
   ....: 1474.5
   ....: Dec-97
   ....: 1040.0
   ....: Dec-82
   ....: 901.0
   ....: Dec-90
   ....: 1129.5
   ....: Dec-98
   ....: 940.5
   ....: Dec-83
   ....: 1573.0
   ....: Dec-91
   ....: 1077.5
   ....: Dec-99
   ....: 1090.5
   ....: Dec-84
   ....: 1055.0
   ....: Dec-92
   ....: 1129.5
   ....: Dec-00
   ....: 1517.5
   ....: Dec-85
   ....: 1488.0
   ....: Dec-93
   ....: 1090.5
   ....: Dec-86
   ....: 1065.5
   ....: Dec-94
   ....: 938.5'''
   ....:

In [23]: import pandas as pd

In [24]: d = d.split('\n')

In [25]: df = pd.DataFrame(zip(d[::2], d[1::2]), columns=['month', 'hdd'])

In [26]: df['hdd'] = df.hdd.astype(float)

In [27]: qd(df.head())

    month    hdd
0  Dec-79  972.5
1  Dec-87 1018.5
2  Dec-95 1199.5
3  Dec-80   1147
4  Dec-88   1155

In [28]: qd(df.describe())

         hdd
count     22
mean  1154.7
std   193.37
min      901
```

(continues on next page)

(continued from previous page)

```
25%   1043.8
50%     1110
75%   1188.6
max     1573
```

Code:

Create an empirical aggregate based on the HDD data.

```
In [29]: hdd = build(f'agg HDD 1 claim dsev {df.hdd.values} fixed'
   ....:             , bs=1/32, log2=16)
   ....:

In [30]: qd(hdd)

       E[X] Est E[X]   Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq      1                           0
Sev  1154.7   1154.7 2.2204e-16 0.16362   0.16362 0.95984     0.95984
Agg  1154.7   1154.7 2.2204e-16 0.16362   0.16362 0.95984     0.95984
log2 = 16, bandwidth = 1/32, validation: not unreasonable.
```

Build the distortion and apply to call options at different strikes. Reproduces Table 2.

```
In [31]: from aggregate.extensions import Formatter

In [32]: d1 = build('distortion w25 wang .25')

In [33]: ans = []

In [34]: strikes = np.arange(1250, 1501, 50)

In [35]: bit = hdd.density_df.query('p_total > 0')[['p_total']]

In [36]: for strike in strikes:
   ....:     ser = bit.groupby(by= lambda x: np.maximum(0, x - strike)).p_total.
↪sum()
   ....:     ans.append(d1.price(ser, kind='both'))
   ....:

In [37]: df = pd.DataFrame(ans, index=strikes,
   ....:                   columns=['bid', 'el', 'ask'])
   ....:

In [38]: df.index.name = 'strike'

In [39]: df['loading'] = (df.ask - df.el) / df.el

In [40]: qd(df.T, float_format=Formatter(dp=2, w=8))

strike      1250     1300     1350     1400     1450     1500
bid        32.01    25.84    19.67    13.51     7.34     2.44
el         47.86    38.77    29.68    20.59    11.50     4.11
ask        68.21    55.45    42.70    29.94    17.18     6.59
loading    42.5%    43.0%    43.8%    45.4%    49.4%    60.1%
```

Gerber: Stop Loss Premiums

Source papers: Gerber [1982].

Concepts: Stop loss and survival functions for Poisson uniform(1,3) aggregate. Claim count 1, 10, and 100.

Source Exhibits: Matches columns labeled exact in Tables 1-6.

Code:

Expected claim count 1.

```
In [41]: gerber1 = build('agg Gerber1 1 claim sev 2 * uniform + 1 poisson'
   ....:                 , bs=1/1024)
   ....:

In [42]: qd(gerber1)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)    Skew(X) Est Skew(X)
X
Freq     1                             1                     1
Sev      2        2           0 0.28868   0.28868 -1.846e-14           0
Agg      2        2 -5.5511e-16  1.0408    1.0408     1.1086      1.1086
log2 = 16, bandwidth = 1/1024, validation: fails sev skew.

In [43]: bit = gerber1.density_df.loc[0:21:2*1024, ['S', 'lev']]

In [44]: bit['stop_loss'] = gerber1.agg_m - bit.lev

In [45]: qd(bit)

                S    lev  stop_loss
loss
0.000     0.63212      0          2
2.000     0.44809 1.1723    0.82773
4.000     0.17095 1.7311    0.26891
6.000    0.048993 1.9282   0.071838
8.000    0.012406 1.9837   0.016267
10.000  0.0027232 1.9967  0.0032535
12.000 0.00052133 1.9994 0.00058146
14.000 8.7627e-05 1.9999 9.3461e-05
16.000 1.3371e-05      2 1.3664e-05
18.000 1.8774e-06      2   1.84e-06
20.000 2.4358e-07      2 2.3017e-07
```

Expected claim count 10.

```
In [46]: gerber10 = build('agg Gerber10 10 claim sev 2 * uniform + 1 poisson'
   ....:                 , bs=1/128)
   ....:

In [47]: qd(gerber10)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)    Skew(X) Est Skew(X)
X
Freq    10                       0.31623              0.31623
Sev      2        2           0 0.28868   0.28868 -1.846e-14           0
Agg     20       20 -8.8818e-16 0.32914   0.32914    0.35056     0.35056
log2 = 16, bandwidth = 1/128, validation: fails sev skew.

In [48]: bit = gerber10.density_df.loc[15:61:5*128, ['S', 'lev']]

In [49]: bit['stop_loss'] = gerber10.agg_m - bit.lev
```

(continues on next page)

(continued from previous page)

```
In [50]: qd(bit)

                  S     lev  stop_loss
loss
15.000    0.76769 14.243     5.7565
20.000    0.47646 17.374     2.6255
25.000    0.21683 19.068    0.93214
30.000   0.072549 19.744    0.25632
35.000   0.018223 19.945   0.055073
40.000  0.0035238 19.991  0.0093834
45.000 0.00053724 19.999  0.0012887
50.000 6.5977e-05     20 0.00014495
55.000 6.6491e-06     20 1.3552e-05
60.000 5.5879e-07     20 1.0673e-06
```

Expected claim count 100.

```
In [51]: gerber100 = build('agg Gerber100 100 claim sev 2 * uniform + 1 poisson'
   ....:                  , bs=1/16)
   ....:

In [52]: qd(gerber100)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)    Skew(X) Est Skew(X)
X
Freq   100                          0.1                   0.1
Sev      2        2           0 0.28868   0.28896 -1.846e-14           0
Agg    200      200 -1.0658e-14 0.10408   0.10409    0.11086     0.11088
log2 = 16, bandwidth = 1/16, validation: fails sev skew.

In [53]: bit = gerber100.density_df.loc[180:301:20*16, ['S', 'lev']]

In [54]: bit['stop_loss'] = gerber100.agg_m - bit.lev

In [55]: qd(bit)

                  S     lev  stop_loss
loss
180.000   0.83089 178.23     21.774
200.000   0.49203 191.69     8.3051
220.000   0.16763 198.04     1.9595
240.000  0.030257 199.74    0.26481
260.000 0.0028586 199.98   0.019934
280.000  0.000144    200 0.00083749
300.000 3.982e-06    200 1.9972e-05
```

Richardson's Deferred Approach to the Limit

Source papers: Embrechts *et al.* [1993], Grübel and Hermesmeier [2000].

Concepts: Given estimators of an unknown quantity $A^* = A(h) + ch^\alpha + O(h^\beta)$ with $\alpha < \beta$. Evaluate at h and h/t. Multiply the estimate at h/t by t^α, subtract the original estimate, and rearrange to get

$$A^* = \frac{t^\alpha A(h/t) - A(h)}{t^\alpha - 1} + O(h^\beta).$$

The truncation error order of magnitude has decreased. The constant c need not be known. Applying this approach to estimate the density as pmf divided by bucket size, f_h/h, Embrechts *et al.* [1993] report the following.

Setup: Poisson(20) exponential aggregate.

Source Exhibits: Figure 1.

The variable `egp3` is treated as the exact answer. It could also be approximated using the series expansion, but this has been shown already, in REF. Set up basic portfolios evaluated at different bucket sizes.

```
In [56]: egp1 = build('agg EGP 20 claims sev expon 1 poisson',
   ....:              bs=1/16, log2=10)
   ....:

In [57]: qd(egp1)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    20                      0.22361            0.22361
Sev      1  0.99984 -0.00016274       1    1.0005        2       1.998
Agg     20   19.997 -0.00016277 0.31623    0.3163  0.47434     0.47429
log2 = 10, bandwidth = 1/16, validation: fails sev mean, agg mean.

In [58]: egp2 = build('agg EGP 20 claims sev expon 1 poisson',
   ....:              bs=1/32, log2=11)
   ....:

In [59]: qd(egp2)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    20                      0.22361            0.22361
Sev      1  0.99996 -4.0689e-05       1    1.0001        2      1.9995
Agg     20   19.999 -4.0713e-05 0.31623   0.31625  0.47434     0.47432
log2 = 11, bandwidth = 1/32, validation: not unreasonable.

In [60]: egp3 = build('agg EGP 20 claims sev expon 1 poisson',
   ....:               log2=16)
   ....:

In [61]: qd(egp3)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    20                      0.22361            0.22361
Sev      1        1 -1.5895e-07       1         1        2           2
Agg     20       20 -1.5895e-07 0.31623   0.31623  0.47434     0.47434
log2 = 16, bandwidth = 1/512, validation: not unreasonable.
```

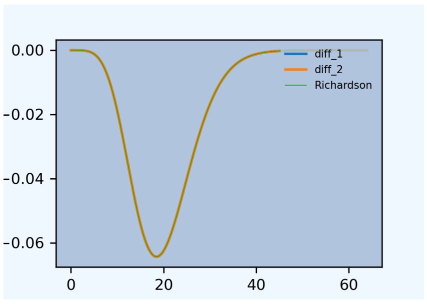
Concatenate and estimated densities from pmf. Compute errors to `egp3`. Compute the Richardson extrapolation. It is indistinguishable from `egp3`. The last table shows cases with the largest errors.

```
In [62]: import pandas as pd

In [63]: df = pd.concat((egp1.density_df.p_total, egp2.density_df.p_total, egp3.
↪density_df.p_total),
   ....:            axis=1, join='inner', keys=[1, 2, 3])
   ....:

In [64]: df[1] = df[1] * 16;                      \
   ....: df[2] = df[2] * 32;                      \
   ....: df[3] = df[3] * (1<<10);                 \
   ....: df['rich'] = (4 * df[2] - df[1]) / 3; \
   ....: df['diff_1'] = df[1] - df[3];            \
   ....: df['diff_2'] = df[2] - df[3];            \
   ....: m = df.diff_2.max() * .9;                \
   ....: ax = df[['diff_1', 'diff_2']].plot(figsize=(3.5, 2.45));       \
   ....: (df['rich'] - df[3]).plot(ax=ax, lw=.5, label='Richardson');
   ....:
```

(continues on next page)

(continued from previous page)

```
In [65]: ax.legend(loc='upper right')
Out[65]: <matplotlib.legend.Legend at 0x7f8089354040>

In [66]: qd(df.query(f'diff_2 > {m}').iloc[::10])

Empty DataFrame
Columns: [1, 2, 3, rich, diff_1, diff_2]
Index: []
```



CHAPTER

THREE

API REFERENCE

3.1 Underwriter Module

3.1.1 Underwriter Class

class aggregate.underwriter.**Underwriter**(*name='Rory'*, *databases=None*, *update=False*, *log2=10*, *debug=False*)

The `Underwriter` class manages the creation of Aggregate and Portfolio objects, and maintains a database of standard Severity (curves) and Aggregate (unit or line level) objects called the knowledge base.

- Handles persistence to and from agg files
- Is interface into program parser
- Handles safe lookup from the knowledge for parser

Objects have a kind and a name. The kind is one of 'sev', 'agg' or 'port'. The name is a string. They have a representation as a program. When the program is interpreted it produces a dictionary spec that can be used to create the object. The static method factory can create any object from the (kind, name, spec, program) quartet, though, strictly, program is not needed.

The underwriter knowledge is stored in a dataframe indexed by kind and name with columns spec and program.

__init__(*name='Rory'*, *databases=None*, *update=False*, *log2=10*, *debug=False*)

Create an underwriter object. The underwriter is the interface to the knowledge base of the aggregate system. It is the interface to the parser and the interpreter, and to the database of curves, portfolios and aggregates.

Parameters

- **name** – name of underwriter. Defaults to Rory, after Rory Cline, the best underwriter I know and a supporter of an analytic approach to underwriting.
- **databases** – name or list of database files to read in on creation. if None: nothing loaded; if 'default' (installed) or 'site' (user, in ~/aggregate/databases) database *.agg files in default or site directory are loaded. If 'all' both default and site databases loaded. A string refers to a single database; an interable of strings is also valid. See *read_database* for search path.
- **update** – if True, update database files with new objects.
- **log2** – log2 of number of buckets in discrete representation. 10 is 1024 buckets.
- **debug** – if True, print debug messages.

_interpreter_work(*iterable*, *debug=False*)

Do all the work for the test, allows input to be marshalled into the tester in different ways. Unlike production interpret_program, runs one line at a time. Each line is preprocessed and then run through a clean parser, and the output analyzed.

Last column, program as input is only changed if the preprocessor changes the program

Returns
DataFrame

build (*program*, *update=None*, *log2=0*, *bs=0*, *recommend_p=0.99999*, *logger_level=None*, ***kwargs*)

Convenience function to make work easy for the user. Intelligent auto updating. Detects discrete distributions and sets `bs = 1`.

`build` method sets loger level to 30 by default.

`__call__` is set equal to `build`.

Parameters

- **program** –
- **update** – build's update
- **log2** – 0 is default: Estimate log2 for discrete and self.log2 for all others. Inupt value over-rides and cancels discrete computation (good for large discrete outcomes where bucket happens to be 1.)
- **bs** –
- **logger_level** – temporary log(ger) level for this build
- **recommend_p** – passed to recommend bucket functions. Increase (closer to 1) for thick tailed distributions.
- **kwargs** – passed to update, e.g., padding. Note force_severity=True is applied automatically

Returns
created object(s)

dir (*pattern=''*)

List all agg databases in site and default directories. If entries is True then read them and return named objects.

Parameters
pattern – glob pattern for filename; .agg is added

factory (*answer*)

Create object of kind from spec, a dictionary. Creating from uw obviously needs the uw, so this is NOT a staticmethod!

Parameters
answer – an Answer class with members kind, name, spec, and program

Returns
creates answer.object

interpret_program (*portfolio_program*)

Preprocess and then parse a program one line at a time. Each output is stored in the Underwriter's knowledge database. No objects are created.

Error handling through parser.

Parameters
portfolio_program –

Returns

interpreter_file (***, *filename=''*, *where=''*)

Run a suite of test programs. For detailed analysis, run_one. filename is a string or Path. If a csv it is read into a dataframe, with the first column used as index. If it is an agg file (e.g. an agg database), it is preprocessed to remove comments and replace nt agg with a space, then split on new lines and converted to a dataframe. Other file formats are rejected.

These methods are called interpreter_… rather than interpret_… because they are for testing and debugging the interpreter, not for actually interpreting anything!

interpreter_line (*program*, *name='one off'*, *debug=True*)

Interpret single line of code in debug mode. name is index of output

interpreter_list (*program_list*)

Interpret elements in a list in debug mode.

static logger_level (*level*)

Convenience function.

Parameters
level –

Returns

more (*regex*)

More information about methods and properties matching regex

qlist (*regex*)

Wrapper for show to just list elements in knowledge that match `regex`. Returns a dataframe.

qshow (*regex*, *tacit=True*)

Wrapper for show to just show (display) elements in knowledge that match `regex`. No reutrn value if tacit, else returns a dataframe.

read_database (*fn*)

read database of curves, aggs and portfolios. These can live in the default directory that is part of the instalation or ~/aggregate/

fn can be a string filename, with or without extension. A .agg extension is added if there is no suffix. Search path:

- in the current dir
- in site_dir (user)
- in default_dir (installation)

Parameters
fn – database file name

run_test_suite ()

Run interpreter on the test suite

safe_lookup (*buildinid*)

Lookup buildinid=kind.name in uw to find expected kind and merge safely into self.arg_dict.

Different from getitem because it splits the item into kind and name and double checks you get the expected kind.

Parameters
buildinid – a string in kind.name format

Returns

show (*regex*, *kind=''*, *plot=True*, *describe=True*, *logger_level=30*, *verbose=False*, ***kwargs*)

Create from knowledge by name or match to name. Optionally plot. Returns the created object plus dataframe with more detailed information. Allows exploration of preloaded databases.

Eg `regex = "A.*[234]` to run examples named A…2, 3 and 4.

See `qshow` for a wrapper that just returns the matches, with no object creation or plotting.

Examples.

```
from aggregate.utilities import pprint
# pretty print all prgrams starting A; no object creation
build.show('^A.*', 'agg', False, False).program.apply(pprint);

# build and plot A..234
ans, df = build.show('^A.*')
```

Parameters

- **`regex`** – for filtering name
- **`kind`** – the kind of object, port, agg, etc.
- **`plot`** – if True, plot (default True)
- **`describe`** – if True, print the describe dataframe
- **`logger_level`** – work silently!
- **`verbose`** – if True, return the dataframe and objects; else no return value
- **`kwargs`** – passed to build for calculation instructions

Returns
dictionary of created objects and DataFrame with info about each.

`property test_suite_file`
Return the test_suite filename, or None if it does not exist

`write`(*portfolio_program*, *log2=0*, *bs=0*, *update=None*, ***kwargs*)
Write a natural language program. Write carries out the following steps.

1. Read in the program and cleans it (e.g. punctuation, parens etc. are removed and ignored, replace ; with new line etc.)
2. Parse line by line to create a dictionary definition of sev, agg or port objects.
3. Replace sev.name, agg.name and port.name references with their objects.
4. If update set, update all created objects.

Sample input:

```
port MY_PORTFOLIO
    agg Line1 20  loss 3 x 2 sev gamma 5 cv 0.30 mixed gamma 0.4
    agg Line2 10  claims 3 x 2 sevgamma 12 cv 0.30 mixed gamma 1.2
    agg Line 3100  premium at 0.4 3 x 2 sev 4 @ lognormal 3 cv 0.8 fixed 1
```

The indents are required if each agg item appears on a new line.

See parser for full language spec! See Aggregate class for many examples.

Parameters

- **`log2`** –
- **`bs`** –
- **`portfolio_program`** –
- **`update`** – override class default
- **`kwargs`** – passed to object's update method if `update==True`

Returns
single created object or dictionary name: object

write_from_file(*file_name, log2=0, bs=0, update=False, **kwargs*)

Read program from file. Delegates to write.

Parameters

- **file_name** –
- **log2** –
- **bs** –
- **update** –
- **kwargs** –

Returns

3.1.2 Other Underwriter functions

3.2 Parser Module

3.2.1 Lexer Class

class aggregate.parser.**UnderwritingLexer**

Implements the Lexer for the agg language.

static preprocess(*program*)

Separate preprocessor step, allowing it to be called separately. Preprocessing involves six steps:

1. Remove // comments, through end of line
2. Remove n in [ ] (vectors) that appear from using `f'{np.linspace(...)}'`
3. Backslash (line continuation) mapped to space
4. nt is replaced with space, supporting the tabbed indented Portfolio layout
5. Split on newlines

Parameters

program –

Returns

3.2.2 Parser Class

class aggregate.parser.**UnderwritingParser**(*safe_lookup_function, debug=False*)

Implements the Parser for the agg language.

Here are testers for the math expressions:

```
from aggregate import build
for t in ['-123', '-2%', '45%', '1e-3%', 'inf', '-inf', 'exp(1)', 'exp(1/2)',
↪'exp(-1)', '-1/8',
          'exp(10)/exp(3**2/2)', '2**10', '50/exp(.3**2/2)', '1/exp(1.9**2 / 2)
↪']:
    a = build(t)
    print(a.name)
    assert float(a.name) == eval(t.replace('%', '/100').replace('exp', 'np.exp
↪').replace('inf', 'np.inf'))
```

To test on the test_suite:

```
df = build.run_test_suite()
assert len(df.query('error != 0')) == 0
```

static enhance_debugfile(*f_out=''*)

Put links in the parser.out debug file, if DEBUGFILE != ‘’.

Parameters

f_out – Path or filename of output. If “” then DEBUGFILE.html used.

Returns

error(*p*)

Default error handling function. This may be subclassed.

3.2.3 Remaining Functions

aggregate.parser.**grammar**(*add_to_doc=False*, *save_to_fn=''*)

Write the grammar at the top of the file as a docstring

To work with multi-rules enter them on one line, like so:

```
@_('builtin_agg PLUS expr', 'builtin_agg MINUS expr')
```

Parameters

- **add_to_doc** – add the grammar to the docstring
- **save_to_fn** – save the grammar to a file

3.3 Distributions Module

3.3.1 Frequency Class

class aggregate.distributions.**Frequency**(*freq_name*, *freq_a*, *freq_b*, *freq_zm*, *freq_p0*)

Manages Frequency distributions: creates moment function and MGF.

- freq_moms(n): returns EN, EN^2 and EN^3 when EN=n
- freq_pgf(n, z): returns the moment generating function applied to z when EN=n

Frequency distributions are either non-mixture types or mixture types.

Non-Mixture Frequency Types

- `fixed`: no parameters
- `bernoulli`: exp_en interpreted as a probability, must be < 1
- `binomial`: Binomial(n/p, p) where p = freq_a, and n = exp_en
- `poisson`: Poisson(n)
- `geometric`: geometric(1/(n + 1)), supported on 0, 1, 2, …
- `logarithmci`: logarithmic(theta), supported on 1, 2, …; theta solved numerically
- `negymana`: Po(n/freq_a) stopped sum of Po(freq_a) freq_a = “eggs per cluster”
- `negbin`: freq_a is the variance multiplier, ratio of variance to mean
- `pascal`:

- `pascal`: (generalized) pascal-poisson distribution, a poisson stopped sum of negative binomial; exp_en gives the overall claim count. freq_a is the CV of the frequency distribution and freq_b is the number of claimants per claim (or claims per occurrence). Hence, the Poisson component has mean exp_en / freq_b and the number of claims per occurrence has mean freq_b. This parameterization may not be ideal(!).

Mixture Frequency Types

These distributions are G-mixed Poisson, so N | G ~ Poisson(n G). They are labelled by the name of the mixing distribution or the common name for the resulting frequency distribution. See Panjer and Willmot or JKK.

In all cases freq_a is the CV of the mixing distribution which corresponds to the asympototic CV of the frequency distribution and of any aggregate when the severity has a variance.

- `gamma`: negative binomial, freq_a = cv of gamma distribution
- `delaporte`: shifted gamma, freq_a = cv of mixing disitribution, freq_b = proportion of certain claims = shift. freq_b must be between 0 and 1.
- `ig`: inverse gaussian, freq_a = cv of mixing distribution
- `sig`: shifted inverse gaussian, freq_a = cv of mixing disitribution, freq_b = proportion of certain claims = shift. freq_b must be between 0 and 1.
- `sichel`: generalized inverse gaussian mixing distribution, freq_a = cv of mixing distribution and freq_b = lambda value. The beta and mu parameters solved to match moments. Note lambda = -0.5 corresponds to inverse gaussian and 0.5 to reciprocal inverse gauusian. Other special cases are available.
- `sichel.gamma`: generalized inverse gaussian mixture where the parameters match the moments of a delaporte distribution with given freq_a and freq_b
- `sichel.ig`: generalized inverse gaussian mixture where the parameters match the moments of a shifted inverse gaussian distribution with given freq_a and freq_b. This parameterization has poor numerical stability and may fail.
- `beta`: beta mixing with freq_a = Cv where beta is supported on the interval [0, freq_b]. This method should be used carefully. It has poor numerical stability and can produce bizzare aggregates when the alpha or beta parameters are < 1 (so there is a mode at 0 or freq_b).

Code proof for Neyman A:

```python
from aggregate import build, qd
mean = 10
eggs_per_cluster = 4
neya = build(f'agg Neya {mean} claims dsev[1] neymana {eggs_per_cluster}')
qd(neya)

po = build(f'agg Po4 {eggs_per_cluster} claims dsev[1] poisson')
po_pmf = po.density_df.query('p_total > 1e-13').p_total

byhand = build(f'agg ByHand {mean / eggs_per_cluster} claims dsev {list(po_pmf.
↪index)} {po_pmf.values} poisson')
qd(byhand)

df = pd.concat((neya.density_df.p_total, byhand.density_df.p_total), axis=1)
df.columns = ['neya', 'byhand']
df['err'] = df.neya - df.byhand
assert df.err.abs().max() < 1e-5
df.head(40)
```

Code proof for Pascal:

```python
from aggregate import build, qd
mean = 10
claims_per_occ =1.24
overall_cv = 1.255
pascal = build(f'agg PascalEg {mean} claims dsev[1] pascal {overall_cv}
```

(continues on next page)

(continued from previous page)

```
↪{claims_per_occ}', log2=16)
qd(pascal)

c = (mean * overall_cv**2 - 1 - claims_per_occ) / claims_per_occ
th = claims_per_occ * c
a = 1 / c
# from form of nb pgf identify r = a and beta = theta, mean is rb, var is␣
↪rb(1+b)
nb = build(f'agg NB {claims_per_occ} claims dsev[1] negbin {th + 1}', log2=16)
nb_pmf = nb.density_df.query('p_total > 1e-13').p_total
qd(nb)

byhand = build(f'agg ByHand {mean / claims_per_occ} claims dsev {list(nb_pmf.
↪index)} {nb_pmf.values} poisson', log2=16)
qd(byhand)

df = pd.concat((pascal.density_df.p_total, byhand.density_df.p_total), axis=1)
df.columns = ['pascal', 'byhand']
df['err'] = df.pascal - df.byhand
assert df.err.abs().max() < 1e-5
df.head(40)
```

Parameters

- **`freq_name`** – name of the frequency distribution, poisson, geometric, etc.
- **`freq_a`** –
- **`freq_b`** –

`__init__`(*freq_name*, *freq_a*, *freq_b*, *freq_zm*, *freq_p0*)

Creates the freq_pgf and moment function:

- moment function(n) returns EN, EN^2, EN^3 when EN=n.
- freq_pgf(n, z) is the freq_pgf evaluated at log(z) when EN=n

Parameters

- **`freq_name`** – name of the frequency distribution, poisson, geometric, etc.
- **`freq_a`** –
- **`freq_b`** –
- **`freq_zm`** – freq_zm True if zero modified, default False
- **`freq_p0`** – modified p0, probability of zero claims

3.3.2 Severity Class

class `aggregate.distributions.`**`Severity`**(*sev_name*, *exp_attachment=None*, *exp_limit=inf*, *sev_mean=0*, *sev_cv=0*, *sev_a=nan*, *sev_b=0*, *sev_loc=0*, *sev_scale=0*, *sev_xs=None*, *sev_ps=None*, *sev_wt=1*, *sev_lb=0*, *sev_ub=inf*, *sev_conditional=True*, *name=''*, *note=''*)

`__init__`(*sev_name*, *exp_attachment=None*, *exp_limit=inf*, *sev_mean=0*, *sev_cv=0*, *sev_a=nan*, *sev_b=0*, *sev_loc=0*, *sev_scale=0*, *sev_xs=None*, *sev_ps=None*, *sev_wt=1*, *sev_lb=0*, *sev_ub=inf*, *sev_conditional=True*, *name=''*, *note=''*)

A continuous random variable, subclasses `scipy.statistics_df.rv_continuous`, adding layer and attachment functionality. It overrides

- `cdf`
- `pdf`
- `isf`
- `ppf`
- `sf`
- `stats`

See scipy.stats continuous rvs for more details about available distributions. The following distributions with two shape parameters are supported:

- Burr (`burr`)
- Generalized Pareto (`genpareto`)
- Generalized gamma (`gengamma`)

It is easy to add others in the code below. With two shape parameters the mean cv input format is not available.

See code in Problems and Solutions to extract distributions from scipy stats by introsepection.

Parameters

- **`sev_name`** – scipy statistics_df continuous distribution | (c|d)histogram cts or discerte | fixed
- **`exp_attachment`** – None if layer is missing, distinct from 0; if a = 0 then losses are conditional on X>a, if a = None then losses are conditional on X>=0
- **`exp_limit`** –
- **`sev_mean`** –
- **`sev_cv`** –
- **`sev_a`** – first shape parameter
- **`sev_b`** – second shape parameter (e.g., beta)
- **`sev_loc`** – scipy.stats location parameter
- **`sev_scale`** – scipy.stats scale parameter
- **`sev_xs`** – for fixed or histogram classes
- **`sev_ps`** –
- **`sev_wt`** – this is not used directly; but it is convenient to pass it in and ignore it because sevs are implicitly created with sev_wt=1.
- **`sev_conditional`** – conditional or unconditional; for severities use conditional
- **`name`** – name of the severity object
- **`note`** – optional note.

`cv_to_shape`(*cv*, *hint=1*)

Create a frozen object of type dist_name with given cv. The lognormal, gamma, inverse gamma and inverse gaussian distributions are solved analytically. Other distributions solved numerically and may be unstable.

Parameters

- **`cv`** –
- **`hint`** –

Returns

mean_to_scale(*shape*, *mean*, *loc=0*)

Adjust the scale to achieved desired mean. Return a frozen instance.

Parameters

- **shape** –
- **mean** –
- **loc** – location parameter (note: location is added to the mean…)

Returns

moms()

Revised moments for Severity class. Trying to compute moments of

X(a,d) = min(d, (X-a)+)

==> E[X(a,d)^n] = int_a^d (x-a)^n f(x) dx + (d-a)^n S(d).

Let x = q(p), F(x) = p, f(x)dx = dp. for 1,2,…n(<=3).

E[X(a,d)^n] = int_{F(a)}^{F(d)} (q(p)-a)^n dp + (d-a)^n S(d)

The base is to compute int_{F(a)}^{F(d)} q(p)^n dp. These are exi below. They are then adjusted to create the moments needed.

Old moments tried to compute int S(x)dx, but that is over a large, non-compact domain and did not work so well. With 0.9.3 old_moms was removed. Old_moms code did this:

```
ex1 = safe_integrate(lambda x: self.fz.sf(x), 1)
ex2 = safe_integrate(lambda x: 2 * (x - self.attachment) * self.fz.sf(x),␣
↪2)
ex3 = safe_integrate(lambda x: 3 * (x - self.attachment) ** 2 * self.fz.
↪sf(x), 3)
```

Test examples

```
def test(mu, sigma, a, y):
    global moms
    import types
    # analytic with no layer attachment
    fz = ss.lognorm(sigma, scale=np.exp(mu))
    tv = np.array([np.exp(k*mu + k * k * sigma**2/2) for k in range(1,4)])

    # old method
    s = agg.Severity('lognorm', sev_a=sigma, sev_scale=np.exp(mu),
                     exp_attachment=a, exp_limit=y)
    est = np.array(s.old_moms())

    # swap out moment routine
    setattr(s, moms.__name__, types.MethodType(moms, s))
    ans = np.array(s.moms())

    # summarize and report
    sg = f'Example: mu={mu}  sigma={sigma}  a={a}  y={y}'
    print(f'{sg}\n{"="*len(sg)}')
    print(pd.DataFrame({'new_ans' : ans, 'old_ans': est,
                        'err': ans/est-1, 'no_la_analytic' : tv}))


test(8.7, .5, 0, np.inf)
test(8.7, 2.5, 0, np.inf)
test(8.7, 2.5, 10e6, 200e6)
```

Example: `mu=8.7`, `sigma=0.5`, `a=0`, `y=inf`

```
        new_ans        old_ans           err  no_la_analytic
0  6.802191e+03  6.802191e+03  3.918843e-11    6.802191e+03
1  5.941160e+07  5.941160e+07  3.161149e-09    5.941160e+07
2  6.662961e+11  6.662961e+11  2.377354e-08    6.662961e+11
```

Example: mu=8.7 sigma=2.5 a=0 y=inf (here the old method failed)

```
        new_ans        old_ans           err  no_la_analytic
0  1.366256e+05  1.366257e+05 -6.942541e-08    1.366257e+05
1  9.663487e+12  1.124575e+11  8.493016e+01    9.669522e+12
2  2.720128e+23  7.597127e+19  3.579469e+03    3.545017e+23
```

Example: mu=8.7 sigma=2.5 a=10000000.0 y=200000000.0

```
        new_ans        old_ans           err  no_la_analytic
0  1.692484e+07  1.692484e+07  2.620126e-14    1.366257e+05
1  1.180294e+15  1.180294e+15  5.242473e-13    9.669522e+12
2  1.538310e+23  1.538310e+23  9.814372e-14    3.545017e+23
```

The numerical issues are very sensitive. Goign for a compromise between speed and accuracy. Only an issue for very thick tailed distributions with no upper limit - not a realistic situation. Here is a tester program for two common cases:

```
logger_level(30) # see what is going on
for sh, dist in zip([1,2,3,4, 3.5,2.5,1.5,.5], ['lognorm']*3 + ['pareto
↪']*4):
    s = Severity(dist, sev_a=sh, sev_scale=1, exp_attachment=0)
    print(dist,sh, s.moms())
    if dist == 'lognorm':
        print('actual', [(n, np.exp(n*n*sh*sh/2)) for n in range(1,4)])
```

Returns

vector of moments. `np.nan` signals unreliable but finite value. `np.inf` is correct, the moment does not exist.

plot (*n=100*, *axd=None*, *figsize=(7.0, 4.9)*, *layout='AB\nCD'*)

Quick plot, updated for 0.9.3 with mosaic and no grid lines. (F(x), x) plot replaced with log density plot.

param n

number of points to plot.

param axd

axis dictionary, if None, create new figure. Must have keys 'A', 'B', 'C', 'D'.

param figsize

(width, height) in inches.

param layout

the subplot_mosaic layout of the figure. Default is 'AB

CD'.

return

3.3.3 Aggregate Class

class aggregate.distributions.**Aggregate**(*name, exp_el=0, exp_premium=0, exp_lr=0, exp_en=0, exp_attachment=None, exp_limit=inf, sev_name='', sev_a=nan, sev_b=0, sev_mean=0, sev_cv=0, sev_loc=0, sev_scale=0, sev_xs=None, sev_ps=None, sev_wt=1, sev_lb=0, sev_ub=inf, sev_conditional=True, sev_pick_attachments=None, sev_pick_losses=None, occ_reins=None, occ_kind='', freq_name='', freq_a=0, freq_b=0, freq_zm=False, freq_p0=nan, agg_reins=None, agg_kind='', note=''*)

__init__(*name, exp_el=0, exp_premium=0, exp_lr=0, exp_en=0, exp_attachment=None, exp_limit=inf, sev_name='', sev_a=nan, sev_b=0, sev_mean=0, sev_cv=0, sev_loc=0, sev_scale=0, sev_xs=None, sev_ps=None, sev_wt=1, sev_lb=0, sev_ub=inf, sev_conditional=True, sev_pick_attachments=None, sev_pick_losses=None, occ_reins=None, occ_kind='', freq_name='', freq_a=0, freq_b=0, freq_zm=False, freq_p0=nan, agg_reins=None, agg_kind='', note=''*)

The *Aggregate* distribution class manages creation and calculation of aggregate distributions. It allows for very flexible creation of Aggregate distributions. Severity can express a limit profile, a mixed severity or both. Mixed frequency types share a mixing distribution across all broadcast terms to ensure an appropriate inter- class correlation.

Parameters

- **name** – name of the aggregate
- **exp_el** – expected loss or vector
- **exp_premium** – premium volume or vector (requires loss ratio)
- **exp_lr** – loss ratio or vector (requires premium)
- **exp_en** – expected claim count per segment (self.n = total claim count)
- **exp_attachment** – occurrence attachment; None indicates no limit clause, which is treated different from an attachment of zero.
- **exp_limit** – occurrence limit
- **sev_name** – severity name or sev.BUILTIN_SEV or meta.var agg or port or similar or vector or matrix
- **sev_a** – scipy stats shape parameter
- **sev_b** – scipy stats shape parameter
- **sev_mean** – average (unlimited) severity
- **sev_cv** – unlimited severity coefficient of variation
- **sev_loc** – scipy stats location parameter
- **sev_scale** – scipy stats scale parameter
- **sev_xs** – xs and ps must be provided if sev_name is (cld)histogram, xs are the bucket break points
- **sev_ps** – ps are the probability densities within each bucket; if buckets equal size no adjustments needed
- **sev_wt** – weight for mixed distribution
- **sev_lb** – lower bound for severity (length of sev_lb must equal length of sev_ub and weights)
- **sev_ub** – upper bound for severity

- **sev_conditional** – if True, severity is conditional, else unconditional.
- **sev_pick_attachments** – if not None, a list of attachment points to define picks
- **sev_pick_losses** – if not None, a list of losses by layer
- **occ_reins** – layers: share po layer xs attach or XXXX
- **occ_kind** – ceded to or net of
- **freq_name** – name of frequency distribution
- **freq_a** – cv of freq dist mixing distribution
- **freq_b** – claims per occurrence (delaporte or sig), scale of beta or lambda (Sichel)
- **freq_zm** – True/False zero modified flag
- **freq_p0** – if freq_zm, provides the modified value of p0; default is nan
- **agg_reins** – layers
- **agg_kind** – ceded to or net of
- **note** – note, enclosed in { }

_apply_reins_work (*reins_list*, *base_density*, *debug=False*)

Actually do the work. Called by apply_reins and reins_audit_df. Only needs self to get limits, which it must guess without q (not computed at this stage). Does not need to know if occ or agg reins, only that the correct base_density is supplied.

Parameters

- **reins_list** –
- **kind** – occ or agg, for debug plotting
- **debug** –

Returns

ceder, netter,

_reins_audit_df_work (*kind='occ'*)

Apply each re layer separately and aggregate loss and other stats.

_repr_html_ ()

For IPython.display

aggregate_error_analysis (*log2*, *bs2_from=None*, ***kwargs*)

Analysis of aggregate error across a range of bucket sizes. If `bs2_from is None` use recommend_bucket plus/mins 3. Note: if distribution does not have a second moment, you must enter bs2_from.

Parameters

- **log2** –
- **bs2_from** – lower bound on bs to use, in log2 terms; estimate using `recommend_bucket` if not input.
- **kwargs** – passed to `update`

apply_agg_reins (*debug=False*, *padding=1*, *tilt_vector=None*)

Apply the entire agg reins structure and save output. For by layer detail create reins_audit_df. Makes sev_density_gross, sev_density_net and sev_density_ceded, and updates sev_density to the requested view.

Not reflected in statistics df.

Returns

apply_distortion(*dist*)

Apply distortion to the aggregate density and append as exag column to density_df. # TODO check consistent with other implementations. :param dist: :return:

apply_occ_reins(*debug=False*)

Apply the entire occ reins structure and save output For by layer detail create reins_audit_df Makes sev_density_gross, sev_density_net and sev_density_ceded, and updates sev_density to the requested view.

Not reflected in statistics df.

Parameters
debug – More verbose.

Returns

approximate(*approx_type='slognorm'*, *output='scipy'*)

Create an approximation to self using method of moments matching.

Compare to Portfolio.approximate which returns a single sev fixed freq agg, this returns a scipy dist by default.

Use case: exam questions with the normal approacimation!

Parameters

- **approx_type** – norm, lognorn, slognorm (shifted lognormal), gamma, sgamma. If 'all' then returns a dictionary of each approx.
- **output** – scipy - frozen scipy.stats continuous rv object; sev_decl - DecL program for severity (to substituate into an agg ; no name) sev_kwargs - dictionary of parameters to create Severity agg_decl - Decl program agg T 1 claim sev_decl fixed any other string - created Aggregate object

Returns
as above.

cdf(*x*, *kind='previous'*)

Return cumulative probability distribution at x using kind interpolation.

2022-10 change: kind introduced; default was linear

Parameters
x – loss size

Returns

cramer_lundberg(*rho*, *cap=0*, *excess=0*, *stop_loss=0*, *kind='index'*, *padding=1*)

Return the Pollaczeck-Khinchine Capital function relating surplus to eventual probability of ruin. Assumes frequency is Poisson.

See Embrechts, Kluppelberg, Mikosch 1.2, page 28 Formula 1.11

TODO: Should return a named tuple.

Parameters

- **rho** – rho = prem / loss - 1 is the margin-to-loss ratio
- **cap** – cap = cap severity at cap, which replaces severity with X | X <= cap
- **excess** – excess = replace severity with X | X > cap (i.e. no shifting)
- **stop_loss** – stop_loss = apply stop loss reinsurance to cap, so X > stop_loss replaced with Pr(X > stop_loss) mass
- **kind** –
- **padding** – for update (the frequency tends to be high, so more padding may be needed)

Returns
ruin vector as pd.Series and function to lookup (no interpolation if kind==index; else interp) capitals

property density_df

Create and return the density_df data frame. A read only property, though if you write d = a.density_df you can obviously edit d. Some duplication of columns (p and p_total) to ensure consistency with Portfolio.

Returns
DataFrame similar to Portfolio.density_df.

property describe

Theoretic and empirical stats. Used in _repr_html_.

discretize (*sev_calc*, *discretization_calc*, *normalize*)

Discretize the severity distributions and weight.

`sev_calc` describes how the severity is discretize, see **`Discretizing the Severity Distribution`_**. The options are discrete=round, forward, backward or moment.

`sev_calc='continuous'` (same as forward, kept for backwards compatibility) is used when you think of the resulting distribution as continuous across the buckets (which we generally don't). The buckets are not shifted and so $Pr(X = b_i) = Pr(b_{i-1} < X \le b_i)$. Note that $b_{i-1} = -bs/2$ is prepended.

We use the discretized distribution as though it is fully discrete and only takes values at the bucket points. Hence, we should use *sev_calc='discrete'*. The buckets are shifted left by half a bucket, so $Pr(X = b_i) = Pr(b_i - b/2 < X \le b_i + b/2)$.

The other wrinkle is the righthand end of the range. If we extend to np.inf then we ensure we have probabilities that sum to 1. But that method introduces a probability mass in the last bucket that is often not desirable (we expect to see a smooth continuous distribution, and we get a mass). The other alternative is to use endpoint = 1 bucket beyond the last, which avoids this problem but can leave the probabilities short. We opt here for the latter and normalize (rescale).

`discretization_calc` controls whether individual probabilities are computed using backward-differences of the survival function or forward differences of the distribution function, or both. The former is most accurate in the right-tail and the latter for the left-tail of the distribution. We are usually concerned with the right-tail, so prefer *survival*. Using *both* takes the greater of the two esimates giving the best of both worlds (underflow makes distribution zero in the right-tail and survival zero in the left tail, so the maximum gives the best estimate) at the expense of computing time.

Sensible defaults: sev_calc=discrete, discretization_calc=survival, normalize=True.

Parameters

- **sev_calc** – discrete=round, forward, backward, or continuous and method becomes discrete otherwise
- **discretization_calc** – survival, distribution or both; in addition the method then becomes survival
- **normalize** – if True, normalize the severity so sum probs = 1. This is generally what you want; but when dealing with thick tailed distributions it can be helpful to turn it off.

Returns

easy_update (*log2=16*, *bs=0*, *recommend_p=0.99999*, *debug=False*, ***kwargs*)

Convenience function, delegates to update_work. Avoids having to pass xs. Also aliased as easy_update for backward compatibility.

Parameters

- **log2** –
- **bs** –

- **recommend_p** – p value passed to recommend_bucket. If > 1 converted to 1 - 10**-p in rec bucket.
- **debug** –
- **kwargs** – passed through to update

Returns

entropy_fit(*n_moments*, *tol=1e-10*, *verbose=False*)

Find the max entropy fit to the aggregate based on n_moments fit. The constant is added (sum of probabilities constraint), for two moments there are n_const = 3 constrains.

Based on discussions with, and R code from, Jon Evans

Run

```
ans = obj.entropy_fit(2)
ans['ans_df'].plot()
```

to compare the fits.

Parameters

- **n_moments** – number of moments to match
- **tol** –
- **verbose** –

Returns

explain_validation()

Explain validation result. Validation computed if needed.

freq_pmf(*log2*)

Return the frequency probability mass function (pmf) computed using 2**log2 buckets. Uses self.en to compute the expected frequency. The *Frequency* does not know the expected claim count, so this is a method of *Aggregate*.

html_info_blob()

Text top of _repr_html_

json()

Write spec to json string.

Returns

limits(*stat='range'*, *kind='linear'*, *zero_mass='include'*)

Suggest sensible plotting limits for kind=range, density, etc., same as Portfolio.

Should optionally return a locator for plots?

Called by ploting routines. Single point of failure!

Must work without q function when not computed (apply_reins_work for occ reins; then use report_ser instead).

Parameters

- **stat** – range or density (for y axis)
- **kind** – linear or log (this is the y-axis, not log of range…that is rarely plotted)
- **zero_mass** – include exclude, for densities

Returns

more (*regex*)

More information about methods and properties matching regex

pdf (*x*)

Probability density function, assuming a continuous approximation of the bucketed density.

Parameters

x –

Returns

picks (*attachments*, *layer_loss_picks*, *debug=False*)

Adjust the computed severity to hit picks targets in layers defined by a. Delegates work to `utilities.picks_work`. See that function for details.

plot (*axd=None*, *xmax=0*, ***kwargs*)

Basic plot with severity and aggregate, linear and log plots and Lee plot.

Parameters

- **xmax** – Enter a "hint" for the xmax scale. E.g., if plotting gross and net you want all on the same scale. Only used on linear scales?
- **axd** –
- **kwargs** – passed to make_mosaic_figure

Returns

pmf (*x*)

Probability mass function, treating aggregate as discrete x must be in the index (?)

pollaczeck_khinchine (*rho*, *cap=0*, *excess=0*, *stop_loss=0*, *kind='index'*, *padding=1*)

Return the Pollaczeck-Khinchine Capital function relating surplus to eventual probability of ruin. Assumes frequency is Poisson.

See Embrechts, Kluppelberg, Mikosch 1.2, page 28 Formula 1.11

TODO: Should return a named tuple.

Parameters

- **rho** – rho = prem / loss - 1 is the margin-to-loss ratio
- **cap** – cap = cap severity at cap, which replaces severity with X | X <= cap
- **excess** – excess = replace severity with X | X > cap (i.e. no shifting)
- **stop_loss** – stop_loss = apply stop loss reinsurance to cap, so X > stop_loss replaced with Pr(X > stop_loss) mass
- **kind** –
- **padding** – for update (the frequency tends to be high, so more padding may be needed)

Returns

ruin vector as pd.Series and function to lookup (no interpolation if kind==index; else interp) capitals

ppf (*p*, *kind='lower'*)

Return quantile function of density_df.p_total.

Definition 2.1 (Quantiles) x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] ≥ α} is the lower α-quantile of X x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] > α} is the upper α-quantile of X.

`kind=='middle'` has been removed.

Parameters

- **p** –

- **kind** – 'lower' or 'upper'.

Returns

property pprogram

pretty print the program

property pprogram_html

pretty print the program to html

price (*p*, *g*, *kind='var'*)

Price using regulatory and pricing g functions, mirroring Portfolio.price. Unlike Portfolio, cannot calibrate. Applying specified Distortions only. If calibration is needed, embed Aggregate in a one-line Portfolio object.

Compute E_price (X wedge E_reg(X)) where E_price uses the pricing distortion and E_reg uses the regulatory distortion.

Regulatory capital distortion is applied on unlimited basis: `reg_g` can be:

- if input < 1 it is a number interpreted as a p value and used to determine VaR capital
- if input > 1 it is a directly input capital number
- d dictionary: Distortion; spec { name = dist name | var, shape=p value a distortion used directly

`pricing_g` is { name = ph|wang and shape=}, if shape (lr or roe not allowed; require calibration).

if ly, must include ro in spec

Parameters

- **p** – a distortion function spec or just a number; if >1 assets, if <1 a prob converted to quantile
- **kind** – var lower upper tvar
- **g** – pricing distortion function

Returns

q (*p*, *kind='lower'*)

Return quantile function of density_df.p_total.

Definition 2.1 (Quantiles) x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] ≥ α} is the lower α-quantile of X x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] > α} is the upper α-quantile of X.

`kind=='middle'` has been removed.

Parameters

- **p** –
- **kind** – 'lower' or 'upper'.

Returns

q_sev (*p*)

Compute quantile of severity distribution, returning element in the index. Very similar code to q, but only lower quantiles.

Parameters

p –

Returns

recommend_bucket(*log2=10*, *p=0.99999*, *verbose=False*)

Recommend a bucket size given 2**N buckets. Not rounded.

For thick tailed distributions need higher p, try p=1-1e-8.

If no second moment, throws a ValueError. You just can't guess in that situation.

Parameters

- **log2** – log2 of number of buckets. log2=10 is default.
- **p** – percentile to use to determine needed range. Default is RECOMMEND_P. if > 1 converted to 1-10**-n.
- **verbose** – print out recommended bucket sizes for 2**n for n in {log2, 16, 13, 10}

Returns

property reinsurance_audit_df

Create and return the _reins_audit_df data frame. Read only property.

Returns

reinsurance_description(*kind='both'*, *width=0*)

Text description of the reinsurance.

Parameters

- **kind** – both, occ, or agg
- **width** – width of text for textwrap.fill; omitted if width==0

property reinsurance_df

Version of density_df tailored to reinsurance. Several cases

- occ program only: **agg_density_**.. is recomputed manually for all three outcomes
- agg program only: **sev_density_**… not set for gcn
- both programs: agg is gcn for the agg program applied to the requested occ output

`_apply_reins_work`

reinsurance_kinds()

Text desciption of kinds of reinsurance applied: None, Occurrence, Aggergate, both.

Returns

property reinsurance_occ_layer_df

How losses are layered by the occurrence reinsurance. Expected loss, CV layer loss, and expected counts to layers.

reinsurance_occ_plot(*axs=None*)

Plots for occurrence reinsurance: occurrence log density and aggregate quantile plot.

property reinsurance_report_df

Create and return a dataframe with the reinsurance report. TODO: sort out the overlap with reinsurance_audit_df (occ and agg) What this function adds is the ceded/net of occ aggregates before application of the agg reinsurance. The pure occ and agg parts are in reinsurance_audit_df.

property report_df

Created on the fly report to audit creation of object. There were some bad choices of columns in audit_df…but it [maybe] embedded in other code…. Eg the use of _1 vs _m for mean is inconsistent.

Returns

rescale (*scale*, *kind='homog'*)

Return a rescaled Aggregate object - used to compute derivatives.

All need to be safe multiplies because of array specification there is an array that is not a numpy array

TODO have parser return numpy arrays not lists!

Parameters

- **scale** – amount of scale
- **kind** – homog of inhomog

Returns

sample (*n*, *replace=True*)

Draw a sample of n items from the aggregate distribution. Wrapper around pd.DataFrame.sample.

property sev

Make exact sf, cdf and pdfs and store in namedtuple for use as sev.cdf etc.

severity_error_analysis (*sev_calc='round'*, *discretization_calc='survival'*, *normalize=True*)

Analysis of severity component errors, uses the current bs in self. Gives detailed, component by component, error analysis of severities. Includes discretization error (bs large relative to mean) and truncation error (tail integral large).

Total S shows the aggregate not severity. Generally about self.n * (1 - sum_p) (per Feller).

sf (*x*)

Return survival function using linear interpolation.

Parameters

x – loss size

Returns

snap (*x*)

Snap value x to the index of density_df, i.e., as a multiple of self.bs.

Parameters

x –

Returns

property spec

Get the dictionary specification, but treat as a read only property

Returns

property spec_ex

All relevant info.

Returns

property statistics

Pandas series of theoretic frequency, severity, and aggregate 1st, 2nd, and 3rd moments. Mean, cv, and skewness.

Returns

tvar (*p*, *kind=''*)

Updated June 2023, 0.13.0

Compute the tail value at risk at threshold p

Definition 2.6 (Tail mean and Expected Shortfall) Assume E[X−] < ∞. Then x¯(α) = TM_α(X) = α^{−1}E[X 1{X≤x(α)}] + x(α) (α − P[X ≤ x(α)]) is α-tail mean at level α the of X. Acerbi and Tasche (2002)

We are interested in the right hand exceedence [?? note > vs ≥] α^{−1}E[X 1{X > x(α)}] + x(α) (P[X ≤ x(α)] − α)

McNeil etc. p66-70 - this follows from def of ES as an integral of the quantile function

q is exact quantile (most of the time) q1 is the smallest index element (bucket multiple) greater than or equal to q

tvar integral is int_p^1 q(s)ds = int_q^infty xf(x)dx = q + int_q^infty S(x)dx we use the last approach. np.trapz approxes the integral. And the missing piece between q and q1 approx as a trapezoid too.

Parameters

- **p** –
- **kind** –

Returns

tvar_sev(*p*)

TVaR of severity - now available for free!

added June 2023

update(*log2=16*, *bs=0*, *recommend_p=0.99999*, *debug=False*, ***kwargs*)

Convenience function, delegates to update_work. Avoids having to pass xs. Also aliased as easy_update for backward compatibility.

Parameters

- **log2** –
- **bs** –
- **recommend_p** – p value passed to recommend_bucket. If > 1 converted to 1 - 10**-p in rec bucket.
- **debug** –
- **kwargs** – passed through to update

Returns

update_work(*xs*, *padding=1*, *tilt_vector=None*, *approximation='exact'*, *sev_calc='discrete'*, *discretization_calc='survival'*, *normalize=True*, *force_severity=False*, *debug=False*)

Compute a discrete approximation to the aggregate density.

See discretize for sev_calc, discretization_calc and normalize.

Quick simple test with log2=13 update took 5.69 ms and _eff took 2.11 ms. So quicker but not an issue unless you are doing many buckets or aggs.

Parameters

- **xs** – range of x values used to discretize
- **padding** – for FFT calculation
- **tilt_vector** – tilt_vector = np.exp(tilt_amount * np.arange(N)), N=2**log2, and tilt_amount * N < 20 recommended
- **approximation** – 'exact' = perform frequency / severity convolution using FFTs. 'slognorm' or 'sgamma' use a shifted lognormal or shifted gamma approximation.
- **sev_calc** – discrete=round, forward, backward, or continuous and method becomes discrete otherwise
- **discretization_calc** – survival, distribution or both; in addition the method then becomes survival

- **normalize** – if True, normalize the severity so sum probs = 1. This is generally what you want; but when dealing with thick tailed distributions it can be helpful to turn it off.
- **force_severity** – make severities even if using approximation, for plotting
- **debug** – run reinsurance in debug model if True.

Returns

property valid

Check if the model appears valid. An answer of True means the model is "not unreasonable". It does not guarantee the model is valid. On the other hand, False means it is definitely suspect. (The interpretation is similar to the null hypothesis in a statistical test). Called and reported automatically by qd for Aggregate objects.

Checks the relative errors (from `self.describe`) for:

- severity mean < eps
- severity cv < 10 * eps
- severity skew < 100 * eps (skewness is more difficult to estimate)
- aggregate mean < eps and < 2 * severity mean relative error (larger values indicate possibility of aliasing and that `bs` is too small).
- aggregate cv < 10 * eps
- aggregate skew < 100 * esp

The default uses eps = 1e-4 relative error. This can be changed by setting the validation_eps variable.

Test only applied for CV and skewness when they are > 0.

Run with logger level 20 (info) for more information on failures.

A Type 1 error (rejecting a valid model) is more likely than Type 2 (failing to reject an invalide one).

Returns
True (interpreted as not unreasonable) if all tests are passed, else False.

var_dict (*p*, *kind='lower'*, *snap=False*)

Make a dictionary of value at risks for the line, mirrors Portfolio.var_dict. Here is just marshals calls to the appropriate var or tvar function.

No epd. Allows the price function to run consistently with Portfolio version.

Example Use:

```
for p, arg in zip([.996, .996, .996, .985, .01], ['var', 'lower', 'upper',
↪'tvar', 'epd']):
    print(port.var_dict(p, arg,  snap=True))
```

Parameters

- **p** –
- **kind** – var (defaults to lower), upper, lower, tvar
- **snap** – snap tvars to index

Returns

3.4 Portfolio Module

3.4.1 Portfolio Class

class aggregate.portfolio.**Portfolio**(*name*, *spec_list*, *uw=None*)

Portfolio creates and manages a portfolio of Aggregate objects each modeling one unit of business. Applications include

- Model a book of insurance
- Model a large account with several sub lines
- Model a reinsurance portfolio or large treaty

__init__(*name*, *spec_list*, *uw=None*)

Create a new `Portfolio` object.

Parameters

- **name** – The name of the portfolio. No spaces or underscores.
- **spec_list** – A list of

1. dictionary: Aggregate object dictionary specifications or
2. Aggregate: An actual aggregate objects or
3. tuple (type, dict) as returned by uw['name'] or
4. string: Names referencing objects in the optionally passed underwriter
5. a single DataFrame: empirical samples (the total column, if present, is ignored); a p_total column is used for probabilities if present

Returns

new `Portfolio` object.

_make_var_tvar(*ser*)

There is no severity version here, so this knows where to store the answer, cf Aggregate version.

_repr_html_()

Updated to mimic Aggregate

accounting_economic_balance_sheet(*a=0*, *p=0*)

story version assumes line 0 = reserves and 1 = prospective….other than that identical

usual a and p rules

add_exa(*df*, *ft_nots=None*)

Use fft to add exeqa_XXX = E(X_i | X=a) to each dist

also add exlea = E(X_i | X <= a) = sum_{x<=a} exa(x)*f(x) where f is for the total ie. self.density_df['exlea_attrit'] = np.cumsum(self.density_df.exa_attrit * self.density_df.p_total) / self.density_df.F

and add exgta = E(X_i | X>a) since E(X) = E(X | X<= a)F(a) + E(X | X>a)S(a) we have exgta = (ex - exlea F) / S

and add the actual expected losses (not theoretical) the empirical amount: self.density_df['e_attrit'] = np.sum(self.density_df.p_attrit * self.density_df.loss)

Mid point adjustment is handled by the example creation routines self.density_df.loss = self.density_df.loss - bs/2

YOU CANNOT HAVE A LINE with a name starting t!!!

See LCA_Examples for original code

Alternative approach to exa: use UC=unconditional versions of exlea and exi_xgta:

- exleaUC = np.cumsum(port.density_df['exeqa_' + col] * port.density_df.p_total) # unconditional
- exixgtaUC =np.cumsum(self.density_df.loc[::-1, 'exeqa_' + col] / self.density_df.loc[::-1, 'loss'] * self.density_df.loc[::-1, 'p_total'])
- exa = exleaUC + exixgtaUC * self.density_df.loss

Parameters

- **df** – data frame to add to. Initially add_exa was only called by update and wrote to self.density_df. But now it is called by gradient too which writes to gradient_df, so we need to pass in this argument
- **ft_nots** – FFTs of the not lines (computed in gradients) so you don't round trip an FFT; gradients needs to recompute all the not lines each time around and it is stilly to do that twice

add_exa_details (*df*, *eta_mu=False*)

From add_exa, details for epd functions and eta_mu flavors.

Note `eta_mu=True` is required for `epd_2` functions.

add_exa_sample (*sample*, *S_calculation='forwards'*)

Computes a version of density_df using sample to compute E[Xi | X]. Then fill in the other ex.... variables using code from Portfolio.add_exa, stripped down to essentials.

If no p_total is given then samples are assumed equally likely. total is added if not given (sum across rows) total is then aligned to the bucket size self.bs using (total/bs).round(0)*bs. The other loss columns are then scaled so they sum to the adjusted total

Next, group by total, sum p_total and average the lines to create E[Xi|X]

This sample is merged into a stripped down density_df. Then the other ex... columns are added. Excludes eta mu columns.

Anticipated use: replace density_df with this, invalidate quantile function and then compute various allocation metrics.

The index on the input sample is ignored.

Formally `extensions.samples.add_exa_sample`.

analysis_collateral (*line*, *c*, *a*, *debug=False*)

E(C(a,c)) expected value of line against not line with collateral c and assets a, c <= a

Parameters

- **line** – line of business with collateral, analyzed against not line
- **c** – collateral, c <= a required; c=0 reproduces exa, c=a reproduces lev
- **a** – assets, assumed less than the max loss (i.e. within the square)
- **debug** –

Returns

analysis_priority (*asset_spec*, *output='df'*)

Create priority analysis report_ser. Can be called multiple times with different `asset_specs` asset_spec either a float used as an epd percentage or a dictionary. Entering an epd percentage generates the dictionary

base = {i: self.epd_2_assets[('not ' + i, 0)](asset_spec) for i in self.line_names}

Parameters

- **asset_spec** – epd

- **output** – df = pandas data frame; html = nice report, markdown = raw markdown text

Returns

analyze_distortion(*dname, dshape=None, dr0=0.025, ddf=5.5, LR=None, ROE=None, p=None, kind='lower', A=None, use_self=False, plot=False, a_max_p=0.99999999, add_comps=True, efficient=True*)

Graphic and summary DataFrame for one distortion showing results that vary by asset level. such as increasing or decreasing cumulative premium.

Characterized by the need to know an asset level, vs. apply_distortion that produced values for all asset levels.

Returns DataFrame with values upto the input asset level…differentiates from apply_distortion graphics that cover the full range.

analyze_pricing will then zoom in and only look at one asset level for micro-dynamics…

Logic of arguments:

```
if data_in == 'self' use self.augmented_df; this implies a distortion self.
↪distortion

else need to build the distortion and apply it
    if dname is a distortion use it
    else built one calibrated to input data

LR/ROE/a/p:
    if p then a=q(p, kind) else p = MESSY
    if LR then P and ROE; if ROE then Q to P to LR
    these are used to calibrate distortion

A newly made distortion is run through apply_distortion with no plot
```

Logic to determine assets similar to calibrate_distortions.

Can pass in a pre-calibrated distortion in dname

Must pass LR or ROE to determine profit

Must pass p or A to determine assets

Output is an *Answer* class object containing

```
Answer(augmented_df=deets, trinity_df=df, distortion=dist, fig1=f1 if plot␣
↪else None,
      fig2=f2 if plot else None, pricing=pricing, exhibit=exhibit, roe_
↪compare=exhibit2,
      audit_df=audit_df)
```

Originally *example_factory*.

example_factory_exhibits included:

do the work to extract the pricing, exhibit and exhibit 2 DataFrames from deets Can also accept an ans object with an augmented_df element (how to call from outside) POINT: re-run exhibits at different p/a thresholds without recalibrating add relevant items to audit_df a = q(p) if both given; if not both given derived as usual

Figures show

Parameters

- **dname** – name of distortion
- **dshape** – if input use dshape and dr0 to make the distortion
- **dr0** –

- **ddf** – r0 and df params for distortion
- **LR** – otherwise use loss ratio and p or a loss ratio
- **ROE** –
- **p** – p value to determine capital.
- **kind** – type of VaR, upper or lower
- **A** –
- **use_self** – if true use self.augmented and self.distortion…else recompute
- **plot** –
- **a_max_p** – percentile to use to set the right hand end of plots
- **add_comps** – add old-fashioned method comparables (included = True as default to make backwards comp.)
- **efficient** –

Returns
various dataframes in an Answer class object

analyze_distortion_add_comps (*ans*, *a_cal*, *p*, *kind*, *ROE*)

make exhibit with comparison to old-fashioned methods: equal risk var/tvar, scaled var/tvar, stand-alone var/tvar, merton perold, co-TVaR. Not all of these methods is additive.

covar method = proportion of variance (additive)

Other methods could be added, e.g. a volatility method?

Note on calculation

Each method computes allocated assets a_i (which it calls Q_i) = Li + Mi + Qi All methods are constant ROEs for capital We have Li in exhibit. Hence:

L = Li P = (Li + ROE ai) / (1 + ROE) = v Li + d ai Q = a - P M = P - L ratios

In most cases, ai is computed directly, e.g. as a scaled proportion of total assets etc.

The covariance method is slightly different.

Mi = vi M, vi = Cov(Xi, X) / Var(X) Pi = Li + Mi Qi = Mi / ROE ai = Pi + Qi

and sum ai = sum Li + sum Mi + sum Qi = L + M + M/ROE = L + M + Q = a as required. To fit it in the same scheme as all other methods we compute qi = Li + Mi + Qi = Li + vi M + vi M / ROE = li + viM(1+1/ROE) = Li + vi M/d, d=ROE/(1+ROE)

Parameters

- **ans** – answer containing dist and augmented_df elements
- **a_cal** –
- **p** –
- **kind** –
- **LR** –
- **ROE** –

Returns
ans Answer object with updated elements

analyze_distortion_plots (*ans*, *dist*, *a_cal*, *p*, *a_max*, *ROE*, *LR*)

Create plots from an analyze_distortion ans class note: this only looks at distortion related items…it doesn't use anything from the comps

Parameters

- **ans** –
- **dist** –
- **a_cal** –
- **p** –
- **a_max** –
- **ROE** –
- **LR** –

Returns

analyze_distortions (*a=0*, *p=0*, *kind='lower'*, *efficient=True*, *augmented_dfs=None*, *regex=''*, *add_comps=True*)

Run analyze_distortion on self.dists

Parameters

- **a** –
- **p** – the percentile of capital that the distortions are calibrated to
- **kind** – var, upper var, tvar, epd
- **efficient** –
- **augmented_dfs** – input pre-computed augmented_dfs (distortions applied)
- **regex** – apply only distortion names matching regex
- **add_comps** – add traditional pricing comps to the answer

Returns

apply_distortion (*dist*, *view='ask'*, *plots=None*, *df_in=None*, *create_augmented=True*, *S_calculation='forwards'*, *efficient=True*)

Apply the distortion, make a copy of density_df and append various columns to create augmented_df.

augmented_df depends on the distortion but only includes variables that work for all asset levels, e.g.

1. marginal loss, lr, roe etc.
2. bottom up totals

Top down total depend on where the "top" is and do not work in general. They are handled in analyze_distortions where you explicitly provide a top.

Does not touch density_df: that is independent of distortions

Optionally produce graphics of results controlled by plots a list containing none or more of:

1. basic: exag_sumparts, exag_total df.exa_total
2. extended: the full original set

Per 0.11.0: no mass at 0 allowed. If you want to use a distortion with mass at 0 you must use a close approximation.

Parameters

- **dist** – agg.Distortion
- **view** – bid or ask price
- **plots** – iterable of plot types
- **df_in** – when called from gradient you want to pass in gradient_df and use that; otherwise use self.density_df
- **create_augmented** (*object*) – store the output in self.augmented_df

- **S_calculation** – if forwards, recompute S summing p_total forwards…this gets the tail right; the old method was backwards, which does not change S
- **efficient** – just compute the bare minimum (T. series, not M. series) and return

Returns
density_df with extra columns appended

apply_distortions (*dist_dict*, *As=None*, *Ps=None*, *kind='lower'*, *efficient=True*)

Apply a list of distortions, summarize pricing and produce graphical output show loss values where $s_ub > S(loss) > s_lb$ by jump

Parameters

- **kind** –
- **dist_dict** – dictionary of Distortion objects
- **As** – input asset levels to consider OR
- **Ps** – input probs (near 1) converted to assets using `self.q()`

Returns

approximate (*approx_type='slognorm'*, *output='scipy'*)

Create an approximation to self using method of moments matching.

Returns a dictionary specification of the portfolio aggregate_project. If updated uses empirical moments, otherwise uses theoretic moments

Parameters

- **approx_type** – slognorm | sgamma | normal
- **output** – return a dict or agg language specification

Returns

as_severity (*limit=inf*, *attachment=0*, *conditional=False*)

Convert portfolio into a severity without recomputing.

Throws an error if self not updated.

Parameters

- **limit** –
- **attachment** –
- **conditional** –

Returns

audits (*kind='all'*, ***kwargs*)

produce audit plots to assess accuracy of outputs.

Currently only exeqa available

Parameters

- **kind** –
- **kwargs** – passed to pandas plot, e.g. set xlim

Returns

best_bucket (*log2=16*, *recommend_p=0.99999*)

Recommend the best bucket. Rounded recommended bucket for log2 points.

TODO: Is this really the best approach?!

Parameters

- **log2** –
- **recommend_p** –

Returns

biv_contour_plot (*fig*, *ax*, *min_loss*, *max_loss*, *jump*, *log=True*, *cmap='Greys'*, *min_density=1e-15*, *levels=30*, *lines=None*, *linecolor='w'*, *colorbar=False*, *normalize=False*, ***kwargs*)

Make contour plot of line A vs line B. Assumes port only has two lines.

Works with an extract density_df.loc[np.arange(min_loss, max_loss, jump), densities] (i.e., jump is the stride). Jump = 100 * bs is not bad…just think about how big the outer product will get!

Parameters

- **fig** –
- **ax** –
- **min_loss** – the density for each line is sampled at min_loss:max_loss:jump
- **max_loss** –
- **jump** –
- **log** –
- **cmap** –
- **min_density** – smallest density to show on underlying log region; not used if log
- **levels** – number of contours or the actual contours if you like
- **lines** – iterable giving specific values of k to plot X+Y=k
- **linecolor** –
- **colorbar** – show color bar
- **normalize** – if true replace Z with Z / sum(Z)
- **kwargs** – passed to contourf (e.g., use for corner_mask=False, vmin,vmax)

Returns

bodoff (***, *p=0.99*, *a=0*)

Determine Bodoff layer asset allocation at asset level a or VaR percentile p, one of which must be provided. Uses formula 14.42 on p. 284 of Pricing Insurance Risk.

Parameters

- **p** – VaR percentile
- **a** – asset level

Returns
Bodoff layer asset allocation by unit

calibrate_blends (*a*, *premium*, *s_values*, *gs_values=None*, *spread_values=None*, *debug=False*)

Input s values and gs values or (market) yield or spread.

A bond with prob s (small) of default is quoted with a yield (to maturity) of r over risk free (e.g., a cat bond spread, or a corporate bond spread over the appropriate Treasury). As a discount bond, the price is v = 1 - d.

B(s) = bid price for 1(U<s) (bond residual value) A(s) = ask price for 1(U<s) (insurance policy)

By no arb A(s) + B(1-s) = 1. By definition g(s) = A(s) (using LI so the particular U doesn't matter. Applied to U = F(X)).

Let v = 1 / (1 + r) and d = 1 - v be the usual theory of interest quantities.

Hence B(1-s) = v = 1 - A(s) = 1 - g(s) and therefore g(s) = 1 - v = d.

The rate of risk discount δ and risk discount factor (nu) ν are defined so that B(1-s) = ν * (1 - s), it is the extra discount applied to the actuarial value that is bid for the bond. It is a function of s. Therefore ν = (1 - d) / (1 - s) = price of bond / actuarial value of payment.

Then, g(s) = 1 - B(1-s) = 1 - ν (1 - s) = ν s + δ.

Thus, if return (i.e., market yield spreads) are input, they convert to discount factors to define g points.

Blend can be defined by extrapolating the last points in a credit curve. If that fails, increase the return on the highest s point and fill in with a constant return to 1.

The ROE on the investment is not the promised return, because the latter does not allow for default.

Set up to be a function of the Portfolio = self. Calibrated to hit premium at asset level a. a must be in the index.

> a = self.pricing_summary.at['a', kind] premium = self.pricing_summary.at['P', kind]

method = extend or roe

Input

blend_d0 is the Book's blend, with roe above the equity point blend_d is calibrated to the same premium as the other distortions

method = extend if f_blend_extend or ccoc

> ccoc = pick and equity point and back into its required roe. Results in a poor fit to the calibration data
>
> extend = extrapolate out the last slope from calibrtion data

Initially tried interpolating the bond yield curve up, but that doesn't work. (The slope is too flat and it interpolates too far. Does not look like a blend distortion.) Now, adding the next point off the credit yield curve as the "equity" point and solving for ROE.

If debug, returns more output, for diagnostics.

`calibrate_distortion` (*name*, *r0=0.0*, *df=[0.0, 0.9]*, *premium_target=0.0*, *roe=0.0*, *assets=0.0*, *p=0.0*, *kind='lower'*, *S_column='S'*, *S_calc='cumsum'*)

Find transform to hit a premium target given assets of `assets`. Fills in the values in `g_spec` and returns params and diagnostics…so you can use it either way…more convenient

Parameters

- **`name`** – name of distortion
- **`r0`** – fixed parameter if applicable
- **`df`** – t-distribution degrees of freedom
- **`premium_target`** – target premium
- **`roe`** – or ROE
- **`assets`** – asset level
- **`p`** –
- **`kind`** –
- **`S_column`** – column of density_df to use for calibration (allows routine to be used in other contexts; if so used must input a premium_target directly. If assets they are used; else max assets used

Returns

`calibrate_distortions` (*LRs=None*, *COCs=None*, *ROEs=None*, *As=None*, *Ps=None*, *kind='lower'*, *r0=0.03*, *df=5.5*, *strict='ordered'*, *S_calc='cumsum'*)

Calibrate assets a to loss ratios LRs and asset levels As (iterables) ro for LY, it $ro/(1+ro)$ corresponds to a minimum rate online

Parameters

- **LRs** – LR or ROEs given
- **ROEs** – ROEs override LRs
- **COCs** – CoCs override LRs, preferred terms to ROE; ROE maintained for backwards compatibility.
- **As** – Assets or probs given
- **Ps** – probability levels for quantiles
- **kind** –
- **r0** – for distortions that have a min ROL
- **df** – for tt
- **strict** – if==’ordered’ then use the book nice ordering else if True only use distortions with no mass at zero, otherwise use anything reasonable for pricing
- **S_calc** –

Returns

cdf (*x*)

distribution function

Parameters
x –

Returns

collapse (*approx_type='slognorm'*)

Returns new Portfolio with the fit

Deprecated…prefer uw.write(self.fit()) to go through the agg language approach.

Parameters
approx_type – slognorm | sgamma

Returns

cotvar (*p*)

Compute the p co-tvar asset allocation using ISA. Asset alloc = exgta = tail expected value, treating TVaR like a pricing variable.

static create_from_sample (*name*, *sample_df*, *bs*, *log2=16*, ***kwargs*)

Create from a multivariate sample, update with bs, execute switcheroo, and return new Portfolio object.

OED: switcheroo, n. a change of position or an exchange, esp. one intended to surprise or deceive; a reversal or turn-about; spec. an unexpected change or ‘twist’ in a story. Also attributive, reversible, reversed.

density_sample (*n=20*, *reg='loss|p_|exeqa_'*)

sample of equally likely points from density_df with interesting columns reg - regex to select the columns

property describe

Theoretic and empirical stats. Used in _repr_html_. Leverage Aggregate object stats; same format

property distortion_df

Nicely formatted version of self.dist_ans (that exhibited several bad choices!).

ROE returned as COC in modern parlance.

property epd_2_assets

Make epd to assets and vice versa Note that the Merton Perold method requies the eta_mu fields, hence set True

equal_risk_epd(*a*)

determine the common epd threshold so sum sa equals a

equal_risk_var_tvar(*p_v*, *p_t*)

solve for equal risk var and tvar: find pv and pt such that sum of individual line VaR/TVaR at pv/pt equals the VaR(p) or TVaR(p_t)

these won't return elements in the index because you have to interpolate hence using kind=middle

explain_validation()

Explain the validation result. Can pass in if already calculated.

static from_DataFrame(*name*, *df*)

create portfolio from pandas dataframe uses columns with appropriate names

Can be fed the agg output of uw.write_test(agg_program)

Parameters

- **name** –
- **df** –

Returns

static from_Excel(*name*, *ffn*, *sheet_name*, ***kwargs*)

read in from Excel

works via a Pandas dataframe; kwargs passed through to pd.read_excel drops all blank columns (mostly for auditing purposes) delegates to from_dataFrame

Parameters

- **name** –
- **ffn** – full file name, including path
- **sheet_name** –
- **kwargs** –

Returns

static from_dict_of_aggs(*prefix*, *agg_dict*, *sub_ports=None*, *uw=None*, *bs=0*, *log2=0*, *padding=2*, ***kwargs*)

Create a portfolio from any iterable with values aggregate code snippets

e.g. agg_dict = {label: agg_snippet }

will create all the portfolios specified in subsets, or all if subsets=='all'

labels for subports are concat of keys in agg_dict, so recommend you use A:, B: etc. as the snippet names. Portfolio names are prefix_[concat element names]

agg_snippet is line agg blah without the tab or newline

Parameters

- **prefix** –
- **agg_dict** –
- **sub_ports** –
- **kwargs** (`bs, log2, padding,`) – passed through to update; update if bs * log2 > 0

Returns

ft (*x*, *tilt=None*)

FT of x with padding and tilt applied

gamma (*a=None*, *p=None*, *kind='lower'*, *compute_stand_alone=False*, *axs=None*, *plot_mode='return'*)

Return the vector gamma_a(x), the conditional layer effectiveness given assets a. Assets specified by percentile level and type (you need a in the index) gamma can be created with no base and no calibration - it does not depend on a distortion. It only depends on total losses.

Returns the total and by layer versions, see "Main Result for Conditional Layer Effectiveness; Piano Diagram" in OneNote

In total gamma_a(x) = E[(a ^ X) / X | X > x] is the average rate of reimbursement for losses above x given capital a. It satisfies int_0^infty gamma_a(x) S(x) dx = E[a ^ X]. Notice the integral is to infinity, regardless of the amount of capital a.

By line gamma_{a,i}(x) = E[E[X_i | X] / X {(X ^ a) / X} 1_{X>x}] / E[{E[X_i | X] / X} 1_{X>x}].

The denominator equals total weights. It is the line-i recovery weighted layer effectiveness. It equals alpha_i(x) S(x).

Now we have

E[X_i(a)] = int_0^infty gamma_{a,i}(x) alpha_i(x) S(x) dx

Note that you need upper and lower q's in aggs now too.

Nov 2020: added arguments for plots; revised axes, separate plots by line

Parameters

- **a** – input a or p and kind as usual
- **p** – asset level percentile
- **kind** – lower or upper
- **compute_stand_alone** – add stand-alone evaluation of gamma
- **axs** – enough axes; only plot if not None
- **plot_mode** – return or linear scale for y axis

Returns

gradient (*epsilon=0.0078125*, *kind='homog'*, *method='forward'*, *distortion=None*, *remove_fuzz=True*, *extra_columns=None*, *do_swap=True*)

Compute the gradient of various quantities relative to a change in the volume of each portfolio component.

Focus is on the quantities used in rate calculations: S, gS, p_total, exa, exag, exi_xgta, exi_xeqq, exeqa, exgta etc.

homog:

inhomog:

Parameters

- **epsilon** – the increment to use; scale is 1+epsilon
- **kind** – homog[ogeneous] or inhomog: homog computes impact of f((1+epsilon)X_i)-f(X_i). Inhomog scales the frequency and recomputes. Note inhomog will have a slight scale issues with E[Severity]
- **method** – forward, central (using epsilon/2) or backwards
- **distortion** – if included derivatives of statistics using the distortion, such as exag are also computed
- **extra_columns** – extra columns to compute dervs of. Note there is virtually no overhead of adding additional columns

- **do_swap** – force the step to replace line with line+epsilon in all not line2's line2!=line1; whether you need this or not depends on what variables you to be differentiated. E.g. if you ask for exa_total only you don't need to swap. But if you want exa_A, exa_B you do, otherwise the d/dA exa_B won't be correct. TODO: replace with code!

Returns
DataFrame of gradients and audit_df in an Answer class

ift (*x*, *tilt=None*)
IFT of x with padding and tilt applied

json (*stream=None*)
write object as json

Parameters
stream –

Returns
stream or text

limits (*stat='range'*, *kind='linear'*, *zero_mass='include'*)
Suggest sensible plotting limits for kind=range, density, .. (same as Aggregate).

Should optionally return a locator for plots?

Called by ploting routines. Single point of failure!

Must work without q function when not computed (apply_reins_work for occ reins…uses report_ser instead).

Parameters

- **stat** – range or density or logy (for log density/survival function…ensure consistency)
- **kind** – linear or log (this is the y-axis, not log of range…that is rarely plotted)
- **zero_mass** – include exclude, for densities

Returns

property line_renamer
plausible defaults for nicer looking names

replaces . or : with space and capitalizes (generally don't use . because it messes with analyze distortion….

leaves : alone

converts X1 to tex

converts XM1 to tex with minus (for reserves)

Returns

make_all (*p=0*, *a=0*, *As=None*)
make all exhibits with sensible defaults if not entered, paid line is selected as the LAST line

make_audit_df (*columns*, *theoretical_stats=None*)
Add or update the audit_df.

merton_perold (*p*, *kind='lower'*)
Compute Merton-Perold capital allocation at VaR(p) capital using VaR as risk measure.

TODO TVaR version of Merton Perold

more (*regex*)
More information about methods and properties matching regex

`multi_premium_capital`(*As*, *keys=None*)

concatenate multiple prem_capital exhibits

`natural_profit_segment_plot`(*ax*, *p*, *line_names*, *colors*, *translations*)

Plot the natural allocations between 1-p and p th percentiles and optionally translate line(s). Works with augmented_df, no input distortion. User must ensure the correct distortion has been applied.

Parameters

- **`ax`** –
- **`p`** –
- **`line_names`** –
- **`colors`** –
- **`translations`** –

Returns

`nice_program`(*wrap_col=90*)

return wrapped version of port program :return:

`pdf`(*x*)

probability density function, assuming a continuous approximation of the bucketed density :param x: :return:

`percentiles`(*pvalues=None*)

report_ser on percentiles and large losses. Uses interpolation, audit_df uses nearest.

Parameters

`pvalues` – optional vector of log values to use. If None sensible defaults provided

Returns

DataFrame of percentiles indexed by line and log

`plot`(*axd=None*, *figsize=(7.0, 2.45)*)

Defualt plot of density, survival functions (linear and log)

Parameters

- **`axd`** – dictionary with plots A and B for density and log density
- **`figsize`** – arguments passed to make_mosaic_figure if no axd

Returns

`pmf`(*x*)

Probability mass function, treating aggregate as discrete x must be in the index (?)

`property pprogram`

pretty print the program to html

`property pprogram_html`

pretty print the program to html

`premium_capital`(*a=0*, *p=0*)

at a if given else p level of capital

pricing story allows two asset levels…handle that with a concat

was premium_capital_detail

price (*p*, *distortion=None*, ***, *allocation='lifted'*, *view='ask'*, *efficient=True*)

Price using regulatory capital and pricing distortion functions.

Compute E_price (X wedge E_reg(X)) where E_price uses the pricing distortion and E_reg uses the regulatory distortion derived from p. p can be input as a probability level converted to assets using *kind*, a level of assets directly (snapped to index).

Regulatory capital distortion is applied on unlimited basis.

Do not attempt to use with a weight_df dataframe from Bounds. For that, use the bounds object logic directly which is much more efficient.

The argument `kind` has been dropped, it is always `'var'`. If that is not the case, convert your asset level to a VaR threshold.

Updated: May 2023 Turns out, really awkward to return the dictionary. In most calls there is just one distortion passed in. The result of the last distortion are reported in ans.price, and there is a new price_dict for the price of each distortion. T

Parameters

- **p** – float; if >1 assets if <1 a prob converted to quantile
- **distortion** – a distortion, list or dictionary (name: dist) of distortions. If None then `self.dists` dictionary is used.
- **allocation** – 'lifted' (default for legacy reasons) or 'linear': treatment in default scenarios. See PIR.
- **view** – bid or ask
- **efficient** – for apply_distortion, lifted only

Returns

PricingResult namedtuple with 'price', 'assets', 'reg_p', 'distortion', 'df'

price_ccoc (*p*, *ccoc*)

Convenience function to price with a constant cost of captial equal `ccoc` at VaR level `p`. Does not invoke a Distortion. Returns standard DataFrame format.

pricing_bounds (*premium*, *a=0*, *p=0*, *n_tps=64*, *kind='tail'*, *slow=False*, *verbose=250*)

Compute the natural allocation premium ranges by unit consistent with total premium at asset level a or p (one of which must be provided).

Unlike typical case with even s values, this is run at the actual S values of the Portfolio.

Visualize:

```
from pandas.plotting import scatter_matrix
ans = port.pricing_bounds(premium, p=0.98)
scatter_matrix(ans.allocs, marker='.', s=5, alpha=1,
               figsize=(10, 10), diagonal='kde' )
```

profit_segment_plot (*ax*, *p*, *line_names*, *dist_name*, *colors=None*, *translations=None*)

Lee diagram for each requested line on a stand-alone basis, loss and risk adj premium using the dist_name distortion. Optionally specify colors, using C{n}. Optionally specify translations applied to each line. Generally, this applies to shift the cat line up by E[non cat] losses to show it overlays the total.

For a Portfolio with line names CAT and NC:

```
port.gross.profit_segment_plot(ax, 0.99999, ['total', 'CAT', 'NC'],
                    'wang', [2,0,1])
```

add translation to cat line:

```
port.gross.profit_segment_plot(ax, 0.99999, ['total', 'CAT', 'NC'],
                    'wang', [2,0,1], [0, E[NC], 0])
```

Parameters

- **ax** – axis on which to render
- **p** – probability level to set upper and lower y axis limits (p and 1-p quantiles)
- **line_names** –
- **dist_name** –
- **colors** –
- **translations** –

Returns

q (*p*, *kind='lower'*)

Return quantile function of density_df.p_total.

Definition 2.1 (Quantiles) x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] ≥ α} is the lower α-quantile of X x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] > α} is the upper α-quantile of X.

`kind=='middle'` has been removed.

Parameters

- **p** –
- **kind** – 'lower' or 'upper'.

Returns

recommend_bucket ()

Data to help estimate a good bucket size.

Returns

remove_fuzz (*df=None*, *eps=0*, *force=False*, *log=''*)

remove fuzz at threshold eps. if not passed use np.finfo(float).eps.

Apply to self.density_df unless df is not None

Only apply if self.remove_fuzz or force :param eps: :param df: apply to dataframe df, default = self.density_df :param force: do regardless of self.remove_fuzz :return:

property renamer

write a sensible renamer for the columns to use thusly

self.density_df.rename(columns=renamer)

write a tex version separately Create once per item…assume lines etc. never change

Returns
dictionary that can be used to rename columns

report (*report_list='quick'*)

Parameters
report_list –

Returns

sample (*n*, *replace=True*, *desired_correlation=None*, *keep_total=True*)

Pull multivariate sample. Apply Iman Conover to induce correlation if required.

sample_compare (*ax=None*)

Compare the sample sum to the independent sum of the marginals.

sample_density_compare (*fuzz=0*)

Compare from density_df

save (*filename=''*, *mode='a'*)

persist to json in filename; if none save to user.json

Parameters

- **filename** –
- **mode** – for file open

Returns

scatter (*marker='.'*, *s=5*, *alpha=1*, *figsize=(10, 10)*, *diagonal='kde'*, ***kwargs*)

Create a scatter plot of marginals against one another, using pandas.plotting scatter_matrix.

Designed for use with samples. Plots exeqa columns

set_a_p (*a*, *p*)

sort out arguments for assets and prob level and make them consistent neither => set defaults a only set p p only set a both do nothing

sf (*x*)

survival function

Parameters
x –

Returns

show_enhanced_exhibits (*fmt='{:.5g}'*)

show all the exhibits created by enhanced_portfolio methods

snap (*x*)

snap value x to the index of density_df

Parameters
x –

Returns

property spec

Get the dictionary specification.

Returns

property spec_ex

All relevant info.

Returns

stand_alone_pricing (*dist*, *p=0*, *kind='var'*, *S_calc='cumsum'*)

Run distortion pricing, use it to determine and ROE and then compute traditional and default pricing, then consolidate the answer

Parameters

- **self** –
- **roe** –
- **p** –
- **kind** –

Returns

from common_scripts.py

stand_alone_pricing_work(*dist*, *p*, *kind*, *roe*, *S_calc='cumsum'*)

Apply dist to the individual lines of self, with capital standard determined by a, p, kind=VaR, TVaR, etc. Return usual data frame with L LR M P PQ Q ROE, and a

Dist can be a distortion, traditional, or defaut pricing modes. For latter two you have to input an ROE. ROE not required for a distortion.

Parameters

- **self** – a portfolio object
- **dist** – "traditional", "default", or a distortion (already calibrated)
- **p** – probability level for assets
- **kind** – var (or lower, upper), tvar or epd (note, p should be small for EPD, to pander, if p is large we use 1-p)
- **roe** – for traditional methods input roe

Returns

exhibit is copied and augmented with the stand-alone statistics

from common_scripts.py

property statistics

Same as statistics df, to be consistent with Aggregate objects :return:

property tm_renamer

rename exa -> TL, exag -> TP etc. :return:

trim_df()

Trim out unwanted columns from density_df

epd used in graphics

Returns

tvar(*p*, *kind=''*)

Compute the tail value at risk at threshold p. Revised June 2023.

Really this function returns ES, CVaR, but in modern terminology this is called TVaR.

Definition 2.6 (Tail mean and Expected Shortfall) Assume E[X−] < ∞. Then x¯(α) = TM_α(X) = α^{−1}E[X 1{X≤x(α)}] + x(α) (α − P[X ≤ x(α)]) is α-tail mean at level α the of X. Acerbi and Tasche (2002)

McNeil etc. p66-70 - this follows from def of ES as an integral of the quantile function

Parameters

- **p** –
- **kind** – No longer neeed as the new method is exact (equals the old

tail) and about 1000x faster. :return:

tvar_threshold(*p*, *kind*)

Find the value pt such that TVaR(pt) = VaR(p) using Bisection method. Will fail if p=0 because signs are the same.

twelve_plot(*fig*, *axs*, *p=0.999*, *p2=0.9999*, *xmax=0*, *ymax2=0*, *biv_log=True*, *legend_font=0*, *contour_scale=10*, *sort_order=None*, *kind='two'*, *cmap='viridis'*)

Twelve-up plot for ASTIN paper and book, by rc index:

Greys for grey color map

11 density 12 log density 13 biv density plot

21 kappa 22 alpha (from alpha beta plot 4) 23 beta (?with alpha)

row 3 = line A, row 4 = line B from alpha beta four 2
1 S, gS, aS, bgS

32 margin 33 shift margin 42 cumul margin 43 natural profit compare

Args

self = portfolio or enhanced portfolio object p control xlim of plots via quantile; used if xmax=0 p2 controls ylim for 33 and 34: stand alone M and natural M; used if ymax2=0 biv_log - bivariate plot on log scale legend_font - fine tune legend font size if necessary sort_order = plot sorts by column and then .iloc[:, sort_order], if None [1,2,0]

from common_scripts.py

`uat` (*As=None*, *Ps=[0.98]*, *LRs=[0.965]*, *r0=0.03*, *num_plots=1*, *verbose=False*)

Reconcile apply_distortion(s) with price and calibrate

Parameters

- **`As`** – Asset levels
- **`Ps`** (`object`) – probability levels used to determine asset levels using quantile function
- **`LRs`** – loss ratios used to determine profitability
- **`r0`** – r0 level for distortions
- **`verbose`** – controls level of output

Returns

`uat_differential` (*line*)

Check the numerical and theoretical derivatives of exa agree for given line

Parameters
`line` –

Returns

`uat_interpolation_functions` (*a0*, *e0*)

Perform quick audit of interpolation functions

Parameters

- **`a0`** – base assets
- **`e0`** – base epd

Returns

`update` (*log2*, *bs*, *approx_freq_ge=100*, *approx_type='slognorm'*, *remove_fuzz=False*, *sev_calc='discrete'*, *discretization_calc='survival'*, *normalize=True*, *padding=1*, *tilt_amount=0*, *trim_df=False*, *add_exa=True*, *force_severity=True*, *recommend_p=0.99999*, *approximation=None*, *debug=False*)

TODO: currently debug doesn't do anything…

Create density_df, performs convolution. optionally adds additional information if `add_exa=True` for allocation and priority analysis

tilting: [@Grubel1999]: Computation of Compound Distributions I: Aliasing Errors and Exponential Tilting (ASTIN 1999) tilt x numbuck < 20 is recommended log. 210 num buckets and max loss from bucket size

Aggregate reinsurance in parser has replaced the aggregate_cession_function (a function of a Portfolio object that adjusts individual line densities; applied after line aggs created but before creating not-lines; actual statistics do not reflect impact.) Agg re by unit is now applied in the Aggregate object.

TODO: consider aggregate covers at the portfolio level…Where in parse - at the top!

Parameters

- **log2** –
- **bs** – bucket size
- **approx_freq_ge** – use method of moments if frequency is larger than `approx_freq_ge`
- **approx_type** – type of method of moments approx to use (slognorm or sgamma)
- **remove_fuzz** – remove machine noise elements from FFT
- **sev_calc** – how to calculate the severity, discrete (point masses as xs) or continuous (uniform between xs points)
- **discretization_calc** – survival or distribution (accurate on right or left tails)
- **normalize** – if true, normalize the severity so sum probs = 1. This is generally what you want; but
- **padding** – for fft 1 = double, 2 = quadruple
- **tilt_amount** – for tiling methodology - see notes on density for suggested parameters
- **epds** – epd points for priority analysis; if None-> sensible defaults
- **trim_df** – remove unnecessary columns from density_df before returning
- **add_exa** – run add_exa to append additional allocation information needed for pricing; if add_exa also add epd info
- **force_severity** – force computation of severities for aggregate components even when approximating
- **recommend_p** – percentile to use for bucket recommendation.
- **approximation** – if not None, use these instructions ('exact')
- **debug** – if True, print debug information

Returns

property valid

Check if the model appears valid. See documentation for Aggregate.valid.

An answer of True does not guarantee the model is valid, but False means it is definitely suspect. (Similar to the null hypothesis in a statistical test). Called and reported automatically by qd for Aggregate objects.

Checks the relative errors (from `self.describe`) for:

- severity mean < eps
- severity cv < 10 * eps
- severity skew < 100 * eps (skewness is more difficult to estimate)
- aggregate mean < eps and < 2 * severity mean relative error (larger values indicate possibility of aliasing and that `bs` is too small).
- aggregate cv < 10 * eps
- aggregate skew < 100 * esp

eps = 1e-3 by default; change in `validation_eps` attribute.

Test only applied for CV and skewness when they are > 0.

Returns
True if all tests are passed, else False.

var (*p*)
value at risk = alias for quantile function

Parameters
p –

Returns

var_dict (*p*, *kind='lower'*, *total='total'*, *snap=False*)
make a dictionary of value at risks for each line and the whole portfolio.

Returns: {line : var(p, kind)} and includes the total as self.name line

if p near 1 and epd uses 1-p.

Example:

for p, arg in zip([.996, .996, .996, .985, .01], ['var', 'lower', 'upper', 'tvar', 'epd']):
print(port.var_dict(p, arg, snap=True))

Parameters

- **p** –
- **kind** – var (defaults to lower), upper, lower, tvar, epd
- **total** – name for total: total=='name' gives total name self.name
- **snap** – snap tvars to index

Returns

3.4.2 Other Portfolio functions

aggregate.portfolio.**check01** (*s*)
add 0 1 at start end

aggregate.portfolio.**convex_points** (*s*, *gs*)
Extract the points that make the convex envelope, including 0 1

Testers:

```
%%sf 1 1 5 5

s_values, gs_values = [.001,.0011, .002,.003, 0.005, .008, .01], [0.002,.02, .
↪03, .035, 0.036, .045, 0.05]
s_values, gs_values = [.001, .002,.003, .009, .011, 1],  [0.02, .03, .035, .05,
↪ 0.05, 1]
s_values, gs_values = [.001, .002,.003, .009, .01, 1],  [0.02, .03, .035, .
↪0351, 0.05, 1]
s_values, gs_values = [0.01, 0.04], [0.03, 0.07]

points = make_array(s_values, gs_values)
ax.plot(points[:, 0], points[:, 1], 'x')

s_values, gs_values = convex_points(s_values, gs_values)
ax.plot(s_values, gs_values, 'r+')

ax.set(xlim=[-0.0025, .1], ylim=[-0.0025, .1])

hull = ConvexHull(points)
for simplex in hull.simplices:
    ax.plot(points[simplex, 0], points[simplex, 1], 'k-', lw=.25)
```

`aggregate.portfolio.`**`make_array`**(*s*, *gs*)

convert to np array and pad with 0 1

`aggregate.portfolio.`**`make_awkward`**(*log2*, *scale=False*)

Decompose a uniform random variable on range(2**log2) into two parts using Eamonn Long's base 4 method.

Usage:

```
awk = make_awkward(16)
awk.density_df.filter(regex='p_[ABt]').cumsum().plot()
awk.density_df.filter(regex='exeqa_[AB]|loss').plot()
```

3.5 Utilities

3.5.1 Moment Aggregator Class

`class` `aggregate.utilities.`**`MomentAggregator`**(*freq_moms=None*)

Purely accumulates moments Used by Portfolio Not frequency aware makes report_ser df and statistics_df

Internal variables agg, sev, freq, tot = running total, 1, 2, 3 = noncentral moments, E(X^k)

Parameters

`freq_moms` – function of one variable returning first three noncentral moments of the underlying frequency distribution

`__init__`(*freq_moms=None*)

`add_f1s`(*f1*, *s1*, *s2*, *s3*)

accumulate new moments defined by f1 and s - fills in f2, f3 based on stored frequency distribution

used by Aggregate

compute agg for the latest values

Parameters

- **`f1`** –
- **`s1`** –
- **`s2`** –
- **`s3`** –

Returns

`add_fs`(*f1*, *f2*, *f3*, *s1*, *s2*, *s3*)

accumulate new moments defined by f and s

used by Portfolio

compute agg for the latest values

Parameters

- **`f1`** –
- **`f2`** –
- **`f3`** –
- **`s1`** –
- **`s2`** –
- **`s3`** –

Returns

add_fs2 (*f1*, *vf*, *s1*, *vs*)

accumulate based on first two moments entered as mean and variance - this is how questions are generally written.

static agg_from_fs (*f1*, *f2*, *f3*, *s1*, *s2*, *s3*)

aggregate_project moments from freq and sev components

Parameters

- **f1** –
- **f2** –
- **f3** –
- **s1** –
- **s2** –
- **s3** –

Returns

static agg_from_fs2 (*f1*, *vf*, *s1*, *vs*)

aggregate_project moments from freq and sev ex and var x

Parameters

- **f1** –
- **vf** –
- **s1** –
- **vs** –

Returns

static column_names ()

list of the moment and statistics_df names for f x s = a

Returns

static cumulate_moments (*m1*, *m2*, *m3*, *n1*, *n2*, *n3*)

Moments of sum of indepdendent variables

Parameters

- **m1** – 1st moment, E(X)
- **m2** – 2nd moment, E(X^2)
- **m3** – 3rd moment, E(X^3)
- **n1** –
- **n2** –
- **n3** –

Returns

get_fsa_stats (*total*, *remix=False*)

get the current f x s = agg statistics_df and moments total = true use total else, current remix = true for total only, re-compute freq statistics_df based on total freq 1

Parameters

- **total** – binary
- **remix** – combine all sevs and recompute the freq moments from total freq

Returns

moments(*mom_type*, *total=True*)

vector of the moments; convenience function

Parameters

- **mom_type** –
- **total** –

Returns

moments_to_mcvsk(*mom_type*, *total=True*)

convert noncentral moments into mean, cv and skewness type = agg | freq | sev | mix delegates work

Parameters

- **mom_type** –
- **total** –

Returns

static static_moments_to_mcvsk(*ex1*, *ex2*, *ex3*)

returns mean, cv and skewness from non-central moments

Parameters

- **ex1** –
- **ex2** –
- **ex3** –

Returns

stats_series(*name*, *limit*, *pvalue*, *remix*)

combine elements into a reporting series handles order, index names etc. in one place

Parameters

- **name** – series name
- **limit** –
- **pvalue** –
- **remix** – called from Aggregate want remix=True to collect mix terms; from Portfolio remix=False

Returns

3.5.2 Moment Wrangler Class

class aggregate.utilities.**MomentWrangler**

Conversion between central, noncentral and factorial moments

Input any one and ask for any translation.

Stores moments as noncentral internally

__init__()

_make_factorial()

add factorial from central

3.5.3 Axis Manager Class

class aggregate.utilities.**AxisManager**(*n*, *figsize=None*, *height=2*, *aspect=1*, *nr=5*)

Manages creation of a grid of axes for plotting. Allows pandas plot and matplotlib to plot to same set of axes.

Always created and managed through axiter_factory function

Parameters

- **n** – number of plots in grid
- **figsize** –
- **height** – height of individual plot
- **aspect** – aspect ratio of individual plot
- **nr** – number of plots per row

__init__(*n*, *figsize=None*, *height=2*, *aspect=1*, *nr=5*)

dimensions()

return dimensions (width and height) of current layout

Returns

static good_grid(*n*, *c=4*)

Good layout for n plots :param n: :return:

grid(*size=0*)

return a block of axes suitable for Pandas if size=0 return all the axes

Parameters
size –

Returns

grid_size(*n*, *subgrid=False*)

appropriate grid size given class parameters

Parameters

- **n** –
- **subgrid** – call is for a subgrid, no special treatment for 6 and 8

Returns

static make_figure(*n*, *aspect=1.5*, ***kwargs*)

make the figure and iterator :param n: :param aspect: :return:

static print_fig(*n*, *aspect=1.5*)

printout code…to insert (TODO copy to clipboard!) :param n: :param aspect: :return:

static size_figure(*r*, *c*, *aspect=1.5*)

reasonable figure size for n plots :param r: :param c: :param aspect: :return:

tidy()

delete unused axes to tidy up a plot

Returns

static tidy_up(*f*, *ax*)

delete unused frames out of a figure :param ax: :return:

3.5.4 Utilities Module

class aggregate.utilities.**Answer**(***kwargs*)

__init__(***kwargs*)

Generic answer wrapping class with plotting

Parameters

kwargs – key=value to wrap

_repr_html_()

List elements

list()

List elements

static nice(*x*)

return a nice rep of x

summary()

just print out the dataframes: horz or vertical as appropriate reasonable styling :return:

class aggregate.utilities.**GCN**(*gross*, *net*, *ceded*)

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new GCN object from a sequence or iterable

_replace(***kwds*)

Return a new GCN object replacing specified fields with new values

ceded

Alias for field number 2

gross

Alias for field number 0

net

Alias for field number 1

class aggregate.utilities.**GreatFormatter**(*sci=True*, *power_range=(-3, 3)*, *offset=True*, *mathText=False*)

__init__(*sci=True*, *power_range=(-3, 3)*, *offset=True*, *mathText=False*)

aggregate.utilities.**approximate_work**(*m*, *cv*, *skew*, *name*, *agg_str*, *note*, *approx_type*, *output*)

Does the work for Portfolio.approximate and Aggregate.approximate. See their documentation.

Parameters

output – scipy - frozen scipy.stats continuous rv object; agg_decl sev_decl - DecL program for severity (to substituate into an agg ; no name) sev_kwargs - dictionary of parameters to create Severity agg_decl - Decl program agg T 1 claim sev_decl fixed any other string - created Aggregate object

aggregate.utilities.**axiter_factory**(*axiter*, *n*, *figsize=None*, *height=2*, *aspect=1*, *nr=5*)

axiter = check_axiter(axiter, …) to allow chaining TODO can this be done in the class somehow?

Parameters

- **axiter** –
- **n** –

- **figsize** –
- **height** –
- **aspect** –
- **nr** –

Returns

`aggregate.utilities.`**`block_iman_conover`**(*unit_losses*, *intra_unit_corrs*, *inter_unit_corr*, *as_frame=False*)

Apply Iman Conover to the unit loss blocks in `unit_losses` with correlation matrices in `intra`.

Then determine the ordering for the unit totals with correlation `inter`.

Re-order each unit, row by row, so that the totals have the desired correlation structure, but leaving the intra unit correlation unchanged.

`unit_losses = [np.arrays or pd.Series]` of losses by subunit within units, without totals

`len(unit_losses) == len(intra_unit corrs)`

For simplicity all normal copula; can add other later if required.

No totals input or output anywhere.

`if as_frame` then a dataframe version returned, for auditing.

Here is some tester code, using great.test_df to make random unit losses. Vary num_units and num_sims as required.

```
def bic_tester(num_units=3, num_sims=10000):
    from aggregate import random_corr_matrix
    # from great import test_df

    # create samples
    R = range(num_units)
    unit_losses = [test_df(num_sims, 3 + i) for i in R]
    totals = [u.sum(1) for u in unit_losses]

    # manual dataframe to check against
    manual = pd.concat(unit_losses + totals, keys=[f'Unit_{i}' for i in R] + [
↪'Total' for i in R], axis=1)

    # for input to method
    unit_losses = [i.to_numpy() for i in unit_losses]
    totals = [i.to_numpy() for i in totals]

    # make corrs
    intra_unit_corrs = [random_corr_matrix(i.shape[1], p=.5, positive=True)␣
↪for i in unit_losses]
    inter_unit_corr = random_corr_matrix(len(totals), p=1, positive=True)

    # apply method
    bic = block_iman_conover(unit_losses, intra_unit_corrs, inter_unit_corr,␣
↪True)

    # extract frame answer, put col names back
    bic.frame.columns = manual.columns
    dm = bic.frame

    # achieved corr
    for i, target in zip(dm.columns.levels[0], intra_unit_corrs + [inter_unit_
↪corr]):
        print(i)
        print((dm[i].corr() - target).abs().max().max())
```

(continues on next page)

(continued from previous page)

```
        # print(dm[i].corr() - target)

    # total corr across subunits
    display(dm.drop(columns=['Total']).corr())

    # total corr across subunits
    display(dm.drop(columns=['Total']).corr())

    return manual, bic, intra_unit_corrs, inter_unit_corr

manual, bic, intra, inter = bic_tester(3, 10000)
```

`aggregate.utilities.`**`easy_formatter`**(*ax*, *which*, *kind*, *places=None*, *power_range=(-3, 3)*, *sep=''*, *unit=''*, *sci=True*, *mathText=False*, *offset=True*)

set which (x, y, b, both) to kind = sci, eng, nice nice = engineering but uses e-3, e-6 etc. see docs for ScalarFormatter and EngFormatter

`aggregate.utilities.`**`estimate_agg_percentile`**(*m*, *cv*, *skew*, *p=0.999*)

Come up with an estimate of the tail of the distribution based on the three parameter fits, ln and gamma

Updated Nov 2022 with a way to estimate p based on lognormal results. How far in the tail you need to go to get an accurate estimate of the mean. See 2_x_approximation_error in the help.

Retain p param for backwards compatibility.

Parameters

- **m** –
- **cv** –
- **skew** –
- **p** – if > 1 converted to 1 - 10**-n

Returns

`aggregate.utilities.`**`explain_validation`**(*rv*)

Explain the validation result rv. Don't over report: if you fail CV don't need to be told you fail Skew too.

`aggregate.utilities.`**`frequency_examples`**(*n*, *ν*, *f*, *κ*, *sichel_case*, *log2*, *xmax=500*)

Illustrate different frequency distributions and frequency moment calculations.

sichel_case = gamma | ig | ‘’

Sample call:

```
df, ans = frequency_examples(n=100, ν=0.45, f=0.5, κ=1.25,
                             sichel_case='', log2=16, xmax=2500)
```

Parameters

- **n** – E(N) = expected claim count
- **ν** – CV(mixing) = asymptotic CV of any compound aggregate whose severity has a second moment
- **f** – proportion of certain claims, 0 <= f < 1, higher f corresponds to greater skewnesss
- **κ** – (kappa) claims per occurrence
- **sichel_case** – gamma, ig or ‘’
- **xmax** –

`aggregate.utilities.`**`friendly`**`(`*df*`)`

Attempt to format df "nicely", in a user-friendly manner. Not designed for big dataframes!

Parameters
`df` –

Returns

`aggregate.utilities.`**`ft`**`(`*z*, *padding*, *tilt*`)`

fft with padding and tilt padding = n makes vector 2^n as long n=1 doubles (default) n=2 quadruples tilt is passed in as the tilting vector or None: easier for the caller to have a single instance

Parameters

- **`z`** –
- **`padding`** – = 1 doubles
- **`tilt`** – vector of tilt values

Returns

`aggregate.utilities.`**`gamma_fit`**`(`*m*, *cv*`)`

`aggregate.utilities.`**`get_fmts`**`(`*df*`)`

reasonable formats for a styler

Parameters
`df` –

Returns

`aggregate.utilities.`**`html_title`**`(`*txt*, *n=1*, *title_case=True*`)`

Parameters

- **`txt`** –
- **`n`** –
- **`title_case`** –

Returns

`aggregate.utilities.`**`ic_noise`**`(`*n*, *d*`)`

Implements steps 1, 2, 3, 4, 5, and 6 This is bottleneck function, therefore cache it It handles the true-up of the random sample to ensure it is exactly independent :param n: row :param d: columns :return:

`aggregate.utilities.`**`ic_rank`**`(`*N*`)`

rankdata function: assign ranks to data, dealing with ties appropriately work by column N is a numpy array

`aggregate.utilities.`**`ic_reorder`**`(`*ranks*, *samples*`)`

put samples into the order determined by ranks array is calibrated to the reference distribution space for the answer

`aggregate.utilities.`**`ic_t_noise`**`(`*n*, *d*, *dof*`)`

as above using multivariate t distribution noise

`aggregate.utilities.`**`ift`**`(`*z*, *padding*, *tilt*`)`

ift that strips out padding and adjusts for tilt

Parameters

- **`z`** –
- **`padding`** –
- **`tilt`** –

Returns

aggregate.utilities.**iman_conover**(*marginals*, *desired_correlation*, *dof=0*, *add_total=True*)

Perform Iman Conover shuffling on input marginals to achieve desired_correlation Desired_correlation must be positive definite and of the correct size. The result has the same rank correlation as a reference sample with the desired linear correlation. Thus, the process relies on linear and rank correlation (for the reference and the input sample) being close.

if dof==0 use normal scores; else you mv t

Sample code:

```
n = 100
df = pd.DataFrame({ f'line_{i}': ss.lognorm(.1 + .2*np.random.rand(),
                scale=10000).rvs(n) for i in range(3)})
desired = np.matrix([[1, -.3, 0], [-.3, 1, .8], [0, .8, 1]])
print(desired)
# check it is a corr matrix
np.linalg.cholesky(desired)

df2 = iman_conover(df, desired)
df2.corr()
df_scatter(df2)
```

Iman Conover Method

Make rank order the same as a reference sample with desired correlation structure.

Reference sample usually chosen as multivariate normal because it is easy and flexible, but you can use **any** reference, e.g. copula based.

The old @Risk software used Iman Conover.

Input: matrix $mathbf X$ of marginals and desired correlation matrix $mathbf S$

1. Make one column of scores $a_i=Phi^{-1}(i/(n+1))$ for $i=1,dots,n$ and rescale to have standard deviation one. 1. Copy the scores $r$ times to make the score matrix $mathbf M$. 1. Randomly permute the entries in each column of $mathbf M$. 1. Compute the correlation matrix $n^{-1}mathbf M'mathbf M$ of the sample scores $mathbf M$. 1. Compute the Choleski decomposition $n^{-1}mathbf M^tmathbf M=mathbf Emathbf E^t$ of the score correlation matrix. 1. Compute $mathbf M' = mathbf M(mathbf E^t)^{-1}$, which is exactly uncorrelated. 1. Compute the Choleski decomposition $mathbf S=mathbf Cmathbf C^t$ of the desired correlation matrix $mathbf S$. 1. Compute $mathbf T=mathbf M'mathbf C^t$. The matrix $mathbf T$ has exactly the desired correlation structure 1. Let $mathbf Y$ be the input matrix $mathbf X$ with each column reordered to have exactly the same **rank ordering** as the corresponding column of $mathbf T$.

Relies on the fact that rank (Spearman) and linear (Pearson) correlation are approximately the same.

aggregate.utilities.**integral_by_doubling**(*func*, *x0*, *err=1e-08*)

Compute $\int_{x_0}^\infty f$ as the sum

$$\int_{x_0}^\infty f = \sum_{n\ge 0} \int_{2^n x_0}^{2^{n+1}x_0} f$$

Caller should check the integral actually converges.

Parameters

- **func** – function to be integrated.
- **x0** – starting x value
- **err** – desired accuracy: stop when incremental integral is <= err.

aggregate.utilities.**introspect**(*ob*)

Discover the non-private methods and properties of an object ob. Returns a pandas DataFrame.

`aggregate.utilities.`**`kaplan_meier`**(*df*, *loss='loss'*, *closed='closed'*)

Compute Kaplan Meier Product limit estimator based on a sample of losses in the dataframe df. For each loss you know the current evaluation in column `loss` and a 0/1 indicator for open/closed in `closed`.

The output dataframe has columns

- index x_i, size of loss
- open - the number of open events of size x_i (open claim with this size)
- closed - the number closed at size x_i
- events - total number of events of size x_i
- n - number at risk at x_i
- s - probability of suriviving past x_i = 1 - closed / n
- pl - cumulative probability of surviving past x_i

See ipython workbook kaplan_meier.ipynb for a check against lifelines and some kaggle data (telco customer churn, https://www.kaggle.com/datasets/blastchar/telco-customer-churn?resource=download https://towardsdatascience.com/introduction-to-survival-analysis-the-kaplan-meier-estimator-94ec5812a97a

Parameters

- **`df`** – dataframe of data
- **`loss`** – column containing loss amount data
- **`closed`** – column indicating if the obervation is a closed claim (1) or open (0)

Returns

dataframe as described above

`aggregate.utilities.`**`kaplan_meier_np`**(*loss*, *closed*)

Feeder to kaplan_meier where loss is np array of loss amounts and closed a same sized array of 0=open, 1=closed indicators.

`aggregate.utilities.`**`knobble_fonts`**(*color=False*)

Not sure we should get into this…

See FigureManager in Great or common.py

https://matplotlib.org/3.1.1/tutorials/intermediate/color_cycle.html

https://matplotlib.org/3.1.1/users/dflt_style_changes.html#colors-in-default-property-cycle

https://matplotlib.org/2.0.2/examples/color/colormaps_reference.html

https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/linestyles.html

https://stackoverflow.com/questions/22408237/named-colors-in-matplotlib

`aggregate.utilities.`**`ln_fit`**(*m*, *cv*)

lognormal parameters

`aggregate.utilities.`**`log_test`**()

” Issue logs at each level

`aggregate.utilities.`**`logarithmic_theta`**(*mean*)

Solve for theta parameter given mean, see JKK p. 288

`aggregate.utilities.`**`logger_level`**(*level=30*, *name='aggregate'*, *verbose=False*)

Code from common.py

Change logger level all loggers containing name Changing for EVERY logger is a really bad idea, you get the endless debug info out of matplotlib find_font, for exapmle.

FWIW, to list all loggers:

```
loggers = [logging.getLogger()]  # get the root logger
loggers = loggers + [logging.getLogger(name) for name in logging.root.manager.
↪loggerDict]
loggers
```

Parameters
level –

Returns

aggregate.utilities.**lognorm_approx**(*ser*)

Lognormal approximation to series, index = loss values, values = density.

aggregate.utilities.**lognorm_lev**(*mu*, *sigma*, *n*, *limit*)

return E(min(X, limit)^n) for lognormal using exact calculation currently only for n=1, 2

Parameters

- **mu** –
- **sigma** –
- **n** –
- **limit** –

Returns

aggregate.utilities.**make_ceder_netter**(*reins_list*, *debug=False*)

Build the netter and ceder functions. It is applied to occ_reins and agg_reins, so should be stand-alone.

The reinsurance functions are piecewise linear functions from 0 to inf which kinks as needed to express the ceded loss as a function of subject (gross) loss.

For example, if `reins_list = [(1, 10, 0), (0.5, 30, 20)]` the program is 10 x 10 and 15 part of 30 x 20 (share=0.5). This requires nodes at 0, 10, 20, 50, and inf.

It is easiest to make the ceder function. Ceded loss at subject loss at x equals the sum of the limits below x plus the cession to the layer in which x lies. The variable `base` keeps track of the layer, `h` of the sum (height) of lower layers. `xs` tracks the knot points, `ys` the values.

```
Break (xs)   Ceded (ys)
      0            0
     10            0
     20           10
     50           25
    inf           25
```

For example:

```
%%sf 1 2

c, n, x, y = make_ceder_netter([(1, 10, 10), (0.5, 30, 20), (.25, np.inf, 50)],
↪ debug=True)

xs = np.linspace(0,250, 251)
ys = c(xs)

ax0.plot(xs, ys)
ax0.plot(xs, xs, ':C7')
ax0.set(title='ceded')

ax1.plot(xs, xs-ys)
ax1.plot(xs, xs, 'C7:')
ax1.set(title='net')
```

Parameters

- **reins_list** – a list of (share of, limit, attach), e.g., (0.5, 3, 2) means 50% share of 3x2 or, equivalently, 1.5 part of 3 x 2. It is better to store share rather than part because it still works if limit == inf.
- **debug** – if True, return layer function xs and ys in addition to the interpolation functions.

Returns

netter and ceder functions; optionally debug information.

`aggregate.utilities.`**`make_corr_matrix`**(*vine_spec*)

Make a correlation matrix from a vine specification, https://en.wikipedia.org/wiki/Vine_copula.

A vine spececification is:

```
row 0: correl of X0...Xn-1 with X0
row 1: correl of X1....Xn-1 with X1 given X0
row 2: correl of X2....Xn-1 with X2 given X0, X1
etc.
```

For example

```
vs = np.array([[1,.2,.2,.2,.2],
               [0,1,.3,.3,.3],
               [0,0,1,.4, .4],
               [0,0,0,1,.5],
               [0,0,0,0,1]])
make_corr_matrix(vs)
```

Key fact is the partial correlation forumula

$$\rho(X,Y|Z) = \frac{(\rho(X,Y) - \rho(X,Z)\rho(Y,Z))}{\sqrt{(1-\rho(X,Z)^2)(1-\rho(Y,Z)^2)}}$$

and therefore

$$\rho(X,Y) = \rho(X,Z)\rho(Y,Z) + \rho(X,Y|Z)\sqrt{((1-\rho(XZ)^2)(1-\rho(YZ)^2))}$$

see https://en.wikipedia.org/wiki/Partial_correlation#Using_recursive_formula.

`aggregate.utilities.`**`make_mosaic_figure`**(*mosaic*, *figsize=None*, *w=3.5*, *h=2.45*, *xfmt='great'*, *yfmt='great'*, *places=None*, *power_range=(-3, 3)*, *sep=''*, *unit=''*, *sci=True*, *mathText=False*, *offset=True*, *return_array=False*)

make mosaic of axes apply format to xy axes default engineering format default w x h per subplot

xfmt='d' for default axis formatting, n=nice, e=engineering, s=scientific, g=great great = engineering with power of three exponents

if return_array then the returns are mor comparable with the old axiter_factory

`aggregate.utilities.`**`make_var_tvar`**(*ser*)

Make var (lower quantile), upper quantile, and tvar functions from a `pd.Series` `ser`, which has index given by losses and p_total values.

`ser` must have a unique monotonic increasing index and all p_totals > 0.

Such a series comes from `a.density_df.query('p_total > 0').p_total`, for example.

Tested using numpy vs pd.Series lookup functions, and this version is much faster. See `var_tvar_test_suite` function below for testers (obviously run before this code was integrated).

Changed in v. 0.13.0

aggregate.utilities.**moms_analytic**(*fz*, *limit*, *attachment*, *n*, *analytic=True*)

Return moments of $E[(X - attachment)^+ \wedge limit]^m$ for m = 1,2,…,n.

To check:

```
# fz = ss.lognorm(1.24)
fz = ss.gamma(6.234, scale=100)
# fz = ss.pareto(3.4234, scale=100, loc=-100)

a1 = moms_analytic(fz, 50, 1234, 3)
a2 = moms_analytic(fz, 50, 1234, 3, False)
a1, a2, a1-a2, (a1-a2) / a1
```

Parameters

- **fz** – frozen scipy.stats instance
- **limit** – double, limit (layer width)
- **attachment** – double, limit
- **n** – int, power
- **analytic** – if True use analytic formula, else numerical integrals

aggregate.utilities.**more**(*self*, *regex*)

Investigate self for matches to the regex. If callable, try calling with no args, else display.

aggregate.utilities.**mu_sigma_from_mean_cv**(*m*, *cv*)

lognormal parameters

aggregate.utilities.**mv**(*x*, *y=None*)

Nice display of mean and variance for Aggregate or Portfolios or entered values.

R style function, no return value.

Parameters

- **x** – Aggregate or Portfolio or float
- **y** – float, if x is a float

Returns

None

aggregate.utilities.**nice_multiple**(*mx*)

Suggest a nice multiple for an axis with scale 0 to mx. Used by the MultipleLocator in discrete plots, where you want an integer multiple. Return 0 to let matplotlib figure the answer. Real issue is stopping multiples like 2.5.

Parameters

mx –

Returns

aggregate.utilities.**parse_note**(*txt*)

Extract kwargs from txt note. Recognizes bs, log2, padding, normalize, recommend_p. CSS format. Split on ; and then look for k=v pairs bs can be entered as 1/32 etc.

Parameters

txt – input text

Return value

dictionary of keyword: typed value

`aggregate.utilities.`**`parse_note_ex`**(*txt*, *log2*, *bs*, *recommend_p*, *kwargs*)

Avoid duplication: this is how the function is used in Underwriter.build.

`aggregate.utilities.`**`partial_e`**(*sev_name*, *fz*, *a*, *n*)

Compute the partial expected value of fz. Computing moments is a bottleneck, so you want analytic computation for the most commonly used types.

Exponential (for mixed exponentials) implemented separate from gamma even though it is a special case.

for k=0,…,n as a np.array

To do: beta? weibull? Burr? invgamma, etc.

Parameters

- **`sev_name`** – scipy.stats name for distribution
- **`fz`** – frozen scipy.stats instance
- **`a`** – double, limit for integral
- **`n`** – int, power

Returns

partial expected value

`aggregate.utilities.`**`partial_e_numeric`**(*fz*, *a*, *n*)

Simple numerical integration version of partial_e for auditing purposes.

`aggregate.utilities.`**`picks_work`**(*attachments*, *layer_loss_picks*, *xs*, *sev_density*, *n=1*, *sf=None*, *debug=False*)

Adjust the layer unconditional expected losses to target. You need int xf(x)dx, but that is fraught when f is a mixed distribution. So we only use the int S version. `fz` was initially a frozen continuous distribution; but adjusted to sf function and dropped need for pdf function.

See notes for how the parts are defined. Notice that:

```
np.allclose(p.layers.v - p.layers.f, p.layers.l - p.layers.e)
```

is true.

Parameters

- **`attachments`** – array of layer attachment points, in ascending order (bottom to top). a[0]>0
- **`layer_loss_picks`** – Target means. If `len(layer_loss_picks)==len(attachments)` then the bottom layer, 0 to a[0], is added. Can be input as unconditional layer severity (i.e., $\mathbb{E}[(X-a)^+\wedge y]$) or as the layer loss pick (i.e., $\mathbb{E}[(X-a)^+\wedge y]'timesn$ where *n* is the number of ground-up (to the insurer) claims. Multiplying and dividing by $S(a)$ shows this equals conditional severity in the layer times the number of claims in the layer.) Actuaries usually estimate the loss pick to the layer in pricing. When called from `Aggregate` the number of ground up claims is known.
- **`en`** – ground-up expected claims. Target is divided by `en`.
- **`xs`** – x values for discretization
- **`sev_density`** – Series of existing severity density from Aggregate.
- **`sf`** – cdf function for the severity distribution.
- **`debug`** – if True, return debug information (layers, density with adjusted probs, audit of layer expected values.

`aggregate.utilities.`**`pprint`**(*txt*)

Simple text version of pprint with line breaking aggregate.utilities.**pprint_ex**(*txt*, *split=0*, *html=False*)

Try to format an agg program. This is difficult because of dfreq and dsev, optional reinsurance, etc. Go for a simple approach of removing unnecessary spacing and removing notes. Notes can be accessed from the spec that is always to hand.

For long programs use split=60 or so, they are split at appropriate points.

Best to use html = True to get colorization.

Parameters

- **txt** – program text input
- **split** – if > 0 split lines at this length
- **html** – if True return html (via pygments) , else return text

aggregate.utilities.**qd**(**argv*, *accuracy=3*, *align=True*, *trim=True*, *ff=None*, ***kwargs*)

Endless quest for a robust display format!

Quick display (qd) a list of objects. Dataframes handled in text with reasonable defaults. For use in documentation.

Param
argv: list of objects to print

Param
accuracy: number of decimal places to display

Param
align: if True, align columns at decimal point (sEngFormatter)

Param
trim: if True, trim trailing zeros (sEngFormatter)

Param
ff: if not None, use this function to format floats, or 'basic', or 'binary'

Kwargs
passed to pd.DataFrame.to_string for dataframes only. e.g., pass dict of formatters by column.

aggregate.utilities.**qdp**(*df*)

Quick describe with nice percentiles and cv for a dataframe.

aggregate.utilities.**random_corr_matrix**(*n*, *p=1*, *positive=False*)

make a random correlation matrix

smaller p results in more extreme correlation 0 < p <= 1

Eg

```
rcm = random_corr_matrix(5, .8)
rcm
np.linalg.cholesky(rcm)
```

positive=True for all entries to be positive

aggregate.utilities.**rearrangement_algorithm_max_VaR**(*df*, *p=0*, *tau=0.001*, *max_n_iter=100*)

Implementation of the Rearragement Algorithm (RA). Determines the worst p-VaR rearrangement of the input variables.

For loss random variables p is usually close to 1.

Embrechts, Paul, Giovanni Puccetti, and Ludger Ruschendorf, 2013, *Model uncertainty and VaR aggregation*, Journal of Banking and Finance 37, 2750–2764.

Worst-Case VaR

Worst value at risk arrangement of marginals.

See Actuarial Review article.

Worst TVaR / Variance arrangement of bivariate data = pair best with worst, second best with second worst, etc., called **countermonotonic** arangement.

More than 2 marginals: can't *make everything negatively correlated with everything else*. If X and Y are negatively correlated and Y and Z are negatively correlated then X and Z will be positively correlated.

Next best attempt: make X countermonotonic to $Y+Z$, Y to $X+Z$ and Z to $X+Y$. Basis of **rearrangement algorithm**.

The Rearrangement Algorithm

1. Randomly permute each column of X, the $N \times d$ matrix of top $1-p$ observations
2. Loop
   - Create a new matrix Y as follows. For column $j = 1, \dots, d$
     - Create a temporary matrix V_j by deleting the jth column of X
     - Create a column vector v whose ith element equals the sum of the elements in the ith row of V_j
     - Set the jth column of Y equal to the jth column of X arranged to have the opposite order to v, i.e. the largest element in the jth column of X is placed in the row of Y corresponding to the smallest element in v, the second largest with second smallest, etc.
   - Compute y, the $N \times 1$ vector with ith element equal to the sum of the elements in the ith row of Y and let $y^* = \min(y)$ be the smallest element of y and compute x^* from X similarly
   - If $y^* - x^* \ge \epsilon$ then set $X = Y$ and repeat the loop
   - If $y^* - x^* < \epsilon$ then break from the loop
3. The arrangement Y is an approximation to the worst :math:` ext{VaR}_p` arrangement of X.

Parameters

- **df** – Input DataFrame containing samples from each marginal. RA will only combine the top 1-p proportion of values from each marginal.
- **p** – If `p==0` assume df has already truncated to the top p values (for each marginal). Otherwise truncate each at the `int(1-p * len(df))`
- **tau** – simulation tolerance
- **max_iter** – maximum number of iterations to attempt

Returns

the top 1-p values of the rearranged DataFrame

`aggregate.utilities.`**`round_bucket`**`(`*bs*`)`

Compute a decent rounded bucket from an input float `bs`.

```
if bs > 1 round to 2, 5, 10, ...

elif bs < 1 find the smallest power of two greater than bs
```

Test cases:

```
test_cases = [1, 1.1, 2, 2.5, 4, 5, 5.5, 8.7, 9.9, 10, 13,
              15, 20, 50, 100, 99, 101, 200, 250, 400, 457,
                500, 750, 1000, 2412, 12323, 57000, 119000,
                1e6, 1e9, 1e12, 1e15, 1e18, 1e21]
for i in test_cases:
    print(i, round_bucket(i))
```

(continues on next page)

(continued from previous page)

```
for i in test_cases:
    print(1/i, round_bucket(1/i))
```

class aggregate.utilities.**sEngFormatter**(*accuracy*, *min_prefix=-6*, *max_prefix=12*, *align=True*, *trim=True*)

Formats float values according to engineering format inside a range of exponents, and standard scientific notation outside.

Uses the same number of significant digits throughout. Optionally aligns at decimal point. That takes up more horizontal space but produces easier to read output.

Based on matplotlib.ticker.EngFormatter and pandas EngFormatter. Converts to scientific notation outside (smaller) range of prefixes. Uses same number of significant digits?

Testers:

```
sef1 = sEngFormatter(accuracy=5, min_prefix=0, max_prefix=12,␣
↪align=True, trim=True)
sef2 = sEngFormatter(accuracy=5, min_prefix=0, max_prefix=12,␣
↪align=False, trim=True)
sef3 = sEngFormatter(accuracy=5, min_prefix=0, max_prefix=12,␣
↪align=True, trim=False)
sef4 = sEngFormatter(accuracy=5, min_prefix=0, max_prefix=12,␣
↪align=False, trim=False)
test = [1.234 * 10**n for n in range(-20,20)]
test = [-i for i in test] + test
for sef in [sef1, sef2, sef3, sef4]:
    print('
```

'.join([sef(i) for i in test]))
print('

')
print('===============') test = [1.234 * 10**n + 3e-16 for n in range(-20,20)] test = [-i for i in test] + test for sef in [sef1, sef2, sef3, sef4]:

print('

'.join([sef(i) for i in test]))

__init__(*accuracy*, *min_prefix=-6*, *max_prefix=12*, *align=True*, *trim=True*)

remove_trailing_zeros(*str_x*, *x*, *dps*)

Remove trailing zeros from a string representation `str_x` of a number `x`. The number of decimal places is `dps`. If the number is in scientific notation then there is no change. Eg with dps == 3, 1.2000 becomes 1.2, 1.000 becomes 1, but 1.200 when x=1.20000001 is unchanged.

aggregate.utilities.**sensible_jump**(*n*, *desired_rows=20*)

return a sensible jump size to output desired_rows given input of n

Parameters

- **n** –
- **desired_rows** –

Returns

aggregate.utilities.**sgamma_fit**(*m*, *cv*, *skew*)

method of moments shifted gamma fit matching given mean, cv and skewness

Parameters

- **m** –
- **cv** –
- **skew** –

Returns

`aggregate.utilities.`**`show_fig`**(*f*, *format='svg'*, ***kwargs*)

Save a figure so it can be placed precisely in output. Used by Underwriter.show to interleaf tables and plots.

Parameters

- **f** – a plt.Figure
- **format** – svg or png
- **kwargs** – passed to savefig

`aggregate.utilities.`**`sln_fit`**(*m*, *cv*, *skew*)

method of moments shifted lognormal fit matching given mean, cv and skewness

Parameters

- **m** –
- **cv** –
- **skew** –

Returns

`aggregate.utilities.`**`style_df`**(*df*)

Style a df similar to pricinginsurancerisk.com styles.

graph background color is B4C3DC and figure (paler) background is F1F8F#

Dropped row lines; bold level0, caption

Parameters
df –

Returns
styled dataframe

`aggregate.utilities.`**`subsets`**(*x*)

all non empty subsets of x, an interable

`aggregate.utilities.`**`suptitle_and_tight`**(*title*, ***kwargs*)

deal with tight layout when there is a suptitle

Parameters
title –

Returns

`aggregate.utilities.`**`test_var_tvar`**(*program*, *bs=0*, *n_ps=1025*, *normalize=False*, *speed_test=False*, *log2=16*)

Run a test suite of programs against new var and tvar functions compared to old aggregate.Aggregate versions

Suggestion:

```
args = [
    ('agg T dfreq [1,2,4,8] dsev [1]', 0, 1025, False),
    ('agg T dfreq [1:8] dsev [2:3]', 0, 1025, False),
    ('agg D dfreq [1:6] dsev [1]', 0, 7, False),
    ('agg T2 10 claims 100 x 0 sev lognorm 10 cv 1 poisson ', 1/64, 1025,␣
↪False),
    ('agg T2 10 claims sev lognorm 10 cv 4 poisson ', 1/16, 1025, False),
```

(continues on next page)

(continued from previous page)

```
    ('agg T2 10 claims sev lognorm 10 cv 4 mixed gamma 1.2 ', 1/8, 1025,␣
↪False),
    ('agg Dice dfreq [1] dsev [1:6]', 0, 7, False)
]

from IPython.display import display, Markdown
for t in args:
    display(Markdown(f'## {t[0]}'))
    bs = t[1]
    test_suite(*t, speed_test=False, log2=16 if bs!= 1/8 else 20)
```

Expected output to show that new functions are 3+ orders of magnitude faster, and agree with the old functions. q and q_upper agree everywhere except the jumps.

`aggregate.utilities.`**`tweedie_convert`**(*, *p=None*, *μ=None*, *σ2=None*, *λ=None*, *α=None*, *β=None*, *m=None*, *cv=None*)

Translate between Tweedie parameters. Input p, μ, σ2 or λ, α, β or λ, m, cv. Remaining parameters are computed and returned in pandas Series.

p, μ, σ2 are the reproductive parameters, μ is the mean and the variance equals σ2 μ^p λ, α, β are the additive parameters; λαβ is the mean, λα(α + 1) β^2 is the variance (α is the gamma shape and β is the scale). λ, m, cv specify the compound Poisson with expected claim count λ and gamma with mean m and cv

In addition, returns p0, the probability mass at 0.

`aggregate.utilities.`**`tweedie_density`**(*x*, *, *p=None*, *μ=None*, *σ2=None*, *λ=None*, *α=None*, *β=None*, *m=None*, *cv=None*)

Exact density of Tweedie distribution from series expansion. Use any parameterization and convert between them with Tweedie convert. Coded for clarity and flexibility not speed. See `tweedie_convert` for parameterization.

`aggregate.utilities.`**`xsden_to_meancv`**(*xs*, *den*)

Compute mean and cv from xs and density.

Consider adding: np.nan_to_num(den)

Note: cannot rely on pd.Series[-1] to work… it depends on the index. xs could be an index :param xs: :param den: :return:

`aggregate.utilities.`**`xsden_to_meancvskew`**(*xs*, *den*)

Compute mean, cv and skewness from xs and density

Consider adding: np.nan_to_num(den)

param xs
param den
return

3.5.5 Constants

`class` `aggregate.constants.`**`Validation`**(*value*)

An enumeration.

3.6 Distortion Module

class aggregate.spectral.**Distortion**(*name, shape, r0=0.0, df=None, col_x='', col_y='', display_name=''*)

Creation and management of distortion functions.

0.9.4: renamed roe to ccoc, but kept creator with roe for backwards compatibility. Oct 2022: renamed wtdtvar to bitvar, but kept …

__init__(*name, shape, r0=0.0, df=None, col_x='', col_y='', display_name=''*)

Create a new distortion.

Tester:

```
ps = np.linspace(0, 1, 201)
for dn in agg.Distortion.available_distortions(True):
    if dn=='clin':
        # shape param must be > 1
        g_dist = agg.Distortion(**{'name': dn, 'shape': 1.25, 'r0': 0.02,
↪'df': 5.5})
    else:
        g_dist = agg.Distortion(**{'name': dn, 'shape': 0.5, 'r0': 0.02,
↪'df': 5.5})
    g_dist.plot()
    g = g_dist.g
    g_inv = g_dist.g_inv

    df = pd.DataFrame({'p': ps, 'gg_inv': g(g_inv(ps)), 'g_invg': g_
↪inv(g(ps)),
    'g': g(ps), 'g_inv': g_inv(ps)})
    print(dn)
    print("errors")
    display(df.query(' abs(gg_inv - g_invg) > 1e-5'))
```

Parameters

- **name** – name of an available distortion, call `Distortion.available_distortions()` for a list
- **shape** – float or [float, float]
- **shape** – shape parameter
- **r0** – risk free or rental rate of interest
- **df** – for convex envelope, dataframe with col_x and col_y used to parameterize or df for t
- **col_x** –
- **col_y** –
- **display_name** – over-ride name, useful for parameterized convex fix distributions

classmethod available_distortions(*pricing=True, strict=True*)

List of the available distortions.

Parameters

- **pricing** – only return list suitable for pricing, excludes tvar and convex
- **strict** – only include those without mass at zero (pricing only)

Returns

static average_distortion(*data*, *display_name*, *n=201*, *el_col='EL'*, *spread_col='Spread'*)

Create average distortion from (s, g(s)) pairs. Each point defines a wtdTVaR with p=s and p=1 points.

Parameters

- **data** –
- **display_name** –
- **n** – number of s values (between 0 and max(EL), 1 is added
- **el_col** – column containing EL
- **spread_col** – column containing Spread

Returns

static bagged_distortion(*data*, *proportion*, *samples*, *display_name=''*)

Make a distortion by bootstrap aggregation (Bagging) resampling, taking the convex envelope, and averaging from data.

Each sample uses proportion of the data.

Data must have two columns: EL and Spread

Parameters

- **data** –
- **proportion** – proportion of data for each sample
- **samples** – number of resamples
- **display_name** – display_name of created distortion

Returns

static convex_example(*source='bond'*)

Example convex distortion using data from https://www.bis.org/publ/qtrpdf/r_qt0312e.pdf.

Parameters

source – bond gives a bond yield curve example, cat gives cat bond / cat reinsurance pricing based example

Returns

static distortions_from_params(*params*, *index*, *r0=0.025*, *df=5.5*, *pricing=True*, *strict=True*)

Make set of dist funs and inverses from params, output of port.calibrate_distortions. params must just have one row for each method and be in the output format of cal_dist.

Called by Portfolio.

Parameters

- **index** –
- **params** – dataframe such that params[index, :] has a [lep, param] etc. pricing=True, strict=True: which distortions to allow df for t distribution
- **r0** – min rol parameters
- **strict** –
- **pricing** –

Returns

g_dual(*x*)

The dual of the distortion function g.

plot (*xs=None*, *n=101*, *both=True*, *ax=None*, *plot_points=True*, *scale='linear'*, *c=None*, *size='small'*, ***kwargs*)

Quick plot of the distortion

Parameters

- **xs** –
- **n** – length of vector is no xs
- **both** – True: plot g and ginv and add decorations, if False just g and no trimmings
- **ax** –
- **plot_points** –
- **scale** – linear as usual or return plots -log(gs) vs -logs and inverts both scales
- **size** – 'small' or 'large' for size of plot, FIG_H or FIG_W. The default is 'small'.
- **kwargs** – passed to matplotlib.plot

Returns

price (*ser*, *a=inf*, *kind='ask'*, *S_calculation='forwards'*)

Compute the bid and ask prices for the distribution determined by `ser` with an asset limit `a`. Index of `ser` need not be equally spaced, so it can be applied to κ. To do this for unit A in portfolio port:

```
ser = port.density_df[['exeqa_A', 'p_total']].\
    set_index('exeqa_A').groupby('exeqa_A').\
    sum()['p_total']
dist.price(ser, port.q(0.99), 'both')
```

Always use `S_calculation='forwards` method to compute S = 1 - cumsum(probs). Computes the price as the integral of gS.

Parameters

- **ser** – pd.Series of is probabilities, indexed by outcomes. Outcomes need not be spaced evenly. `ser` is usually a probability column from `density_df`.
- **kind** – is "ask", "bid", or "both", giving the pricing view.
- **a** – asset level. `ser` is truncated at `a`.

price2 (*ser*, *a=None*, *S_calculation='forwards'*)

Compute the bid and ask prices for the distribution determined by `ser` with an asset limits given by values of `ser`. Index of `ser` need not be equally spaced, so it can be applied to κ. To do this for unit A in portfolio `port`:

```
ser = port.density_df[['exeqa_A', 'p_total']].\
    set_index('exeqa_A').groupby('exeqa_A').\
    sum()['p_total']
dist.price(ser, port.q(0.99))
```

Parameters

ser – pd.Series of is probabilities, indexed by outcomes. Outcomes must be spaced evenly. `ser` is usually a probability column from `density_df`.

static s_gs_distortion (*s*, *gs*, *display_name=''*)

Make a convex envelope distortion from {s, g(s)} points.

Parameters

- **s** – iterable (can be converted into numpy.array
- **gs** –

- **display_name** –

Returns

classmethod test(*r0=0.035*, *df=[0.0, 0.9]*)

Tester: make some nice plots of available distortions.

Returns

static wtd_tvar(*ps*, *wts*, *display_name=''*, *details=False*)

A careful version of wtd tvar with knots at ps and wts.

Parameters

- **ps** –
- **wts** –
- **display_name** –
- **details** –

Returns

aggregate.spectral.**approx_ccoc**(*roe*, *eps=1e-14*, *display_name=None*)

Create a continuous approximation to the CCoC distortion with return roe. Helpful utility function for creating a distortion.

Parameters

- **roe** – return on equity
- **eps** – small number to avoid mass at zero

aggregate.spectral.**tvar_weights**(*d*)

Return tvar weight function for a distortion d. Use np.gradient to differentiate g' but adjust for certain distortions. The returned function expects a numpy array of p values.

Param

d distortion

3.7 Bounds Module

class aggregate.bounds.**Bounds**(*distribution_spec*)

Implement IME 2022 pricing bounds methodology.

Typical usage: First, create a Portfolio or Aggregate object a. Then

```
bd = cd.Bounds(a)
bd.tvar_cloud('line', premium=, a=, n_tps=, s=, kind=)
p_star = bd.p_star('line', premium)
bd.cloud_view(axes, ...)
```

Parameters

distribution_spec – A Portfolio or Portfolio.density_df dataframe or pd.Series (must have loss as index) If DataFrame or Series values interpreted as desnsity, sum to 1. F, S, exgta all computed using Portfolio methdology If DataFrame line –> p_{line}

__init__(*distribution_spec*)

cloud_view (***, *axs=None*, *n_resamples=0*, *scale='linear'*, *alpha=0.05*, *pricing=True*, *distortions='ordered'*, *title=''*, *lim=(-0.025, 1.025)*, *check=False*, *add_average=True*)

Visualize the distortion cloud with n_resamples. Execute after computing weights.

Parameters

- **axs** –
- **n_resamples** – if random sample
- **scale** – linear or return
- **alpha** – opacity
- **pricing** – restrict to p_max = 0, ensuring g(s)<1 when s<1
- **distortions** – 'ordered' shows the usual calibrated distortions, else list of dicts name:distortion.
- **title** – optional title (applied to all plots)
- **lim** – axis limits
- **check** – construct and plot Distortions to check working ; reduces n_resamples to 5

Returns

compute_weight (*premium*, *p0*, *p1*, *b=inf*, *kind='interp'*)

compute the weight for a single TVaR p0 < p1 value pair

Parameters

- **line** –
- **premium** –
- **tp** –
- **b** –

Returns

compute_weights (*line*, *premium*, *n_tps*, *b=inf*, *kind='interp'*)

Compute the weights of the extreme distortions

Applied to min(line, b) (allows to work for net)

Note: independent of the asset level

Parameters

- **line** – within port, or total
- **premium** – target premium for the line
- **n_tps** – number of tvar p points (tps)number of tvar p points (tps)number of tvar p points (tps)number of tvar p points (tps).
- **b** – loss bound: compute weights for min(line, b); generally used for net losses only.

Returns

distortion (*pl*, *pu*)

Return the BiTVaR with probabilities pl and pu

make_ps (*n*, *mode*)

If add_one then you want n = 2**m + 1 to ensure nicely spaced points.

Mode: making s points (always uniform) or tvar p points (use t_mode). self.t_mode == 'u': make uniform s points against which to evaluate g from 0 to 1 self.t_mode == 'gl': make Gauss-Legndre p points at which TVaRs are evaluated from 0 inclusive to 1 exclusive with more around 1

Parameters
n –

Returns

make_tvar_function (*line*, *b=inf*)

Change in 0.14.0 with new tvar methodology, this function reflects the b limit, it is the TVaR of min(X, b)

Make unlimited TVaR function for line, `self.tvar_unlimited_function`, and set self.Fb.

- Portfolio or Aggregate: get from object
- DataFrame: make from p_{line} column
- Series: make from Series

In the last two cases, uses aggregate.utilties.make_var_tvar_function.

Includes determining sup and putting in value for zero. If sup is largest value in index, sup set to inf.

You generally want to apply with a limit, call `self.tvar_with_bounds`.

Parameters

- **line** – only used for portfolio objects, to specify line (or 'total')
- **b** – bound on the losses, e.g., to model limited liability insurer

Returns

p_star (*line*, *premium*, *b=inf*, *kind='interp'*)

Compute p* so TVaR @ p* of min(X, b) = premium

In this case the cap b has an impact (think of integrating q(p) over p to 1, q is impacted by b)

premium <= b is required (no rip off condition)

If b < inf then must solve TVaR(p) - (1 - F(b)) / (1 - p)[TVaR(F(b)) - b] = premium Let k = (1 - F(b)) [TVaR(F(b)) - b], so solving

f(p) = TVaR(p) - k / (1 - p) - premium == 0

using NR

Parameters

- **line** –
- **premium** – target premium
- **b** – bound
- **kind** – now ignored

Returns

ped_distortion (*n*, *solver='rs'*)

make the approximating distortion from the first n Principal Extreme Distortions (PED)s using rs or ip solutions

Parameters
n –

Returns

principal_extreme_distortion_analysis (*gs*, *pricing=False*)

Find the principal extreme distortion analysis to solve for gs = g(s), s=self.cloud_df.index

Assumes that tvar_cloud has been called and that cloud_df exists len(gs) = len(cloud_df)

E.g., call

b = Bounds(port) b.t_mode = 'u' # set premium and asset level a b.tvar_cloud('total', premium, a) # make gs b.principal_extreme_distortion_analysis(gs)

Parameters

- **gs** – either g(s) evaluated on s = cloud_df.index or the name of a calibrated distortion in distribution_spec.dists (created by a call to calibrate_distortions)
- **pricing** – if try, try just using pricing distortions

Returns

quick_price (*distortion*, *a*)

price total to assets a using distortion

requires distribution_spec has a density_df dataframe with a p_total or p_total

TODO: add ability to price other lines :param distortion: :param a: :return:

tvar_array (*line*, *n_tps=257*, *b=inf*, *kind='interp'*)

Compute tvars at n equally spaced points, tps.

Parameters

- **line** –
- **n_tps** – number of tvar p points, default 257 (assuming add-one mode)
- **b** – cap on losses applied before computing TVaRs (e.g., adjust losses for finite assets b). Use np.inf for unlimited losses.
- **kind** – now ignored.

Returns

tvar_cloud (*line*, *premium*, *a*, *n_tps*, *s*, *kind='interp'*)

weight down tvar functions to the extremal convex measures

asset level a acts like an agg stop on what is being priced, i.e. we are working with min(X, a)

Parameters

- **line** –
- **premium** –
- **a** –
- **n_tps** –
- **s** –
- **b** – bound, applies to min(line, b)

Returns

tvar_hinges (*s*)

make the tvar hinge functions by evaluating each tvar_p(s) = min(1, s/(1-p) for p in tps, at EP points s

all arguments in [0,1] x [0,1]

Parameters

s –

Returns

tvar_with_bound(*p*, *b=inf*, *kind='interp'*)

Compute tvar taking bound into account. Assumes tvar_unfunction setup.

Warning: b must equal the b used when calibrated. The issue is computing F which varies with the type of underlying portfolio. This is fragile. Added storing b and checking equal. For backwards comp. need to keep b argument

Parameters

- **p** –
- **b** –
- **kind** – now ignored

Returns

class aggregate.bounds.**Bounds**(*distribution_spec*)

Implement IME 2022 pricing bounds methodology.

Typical usage: First, create a Portfolio or Aggregate object a. Then

```
bd = cd.Bounds(a)
bd.tvar_cloud('line', premium=, a=, n_tps=, s=, kind=)
p_star = bd.p_star('line', premium)
bd.cloud_view(axes, ...)
```

Parameters

distribution_spec – A Portfolio or Portfolio.density_df dataframe or pd.Series (must have loss as index) If DataFrame or Series values interpreted as desnsity, sum to 1. F, S, exgta all computed using Portfolio methdology If DataFrame line –> p_{line}

cloud_view(**, axs=None, n_resamples=0, scale='linear', alpha=0.05, pricing=True, distortions='ordered', title='', lim=(-0.025, 1.025), check=False, add_average=True*)

Visualize the distortion cloud with n_resamples. Execute after computing weights.

Parameters

- **axs** –
- **n_resamples** – if random sample
- **scale** – linear or return
- **alpha** – opacity
- **pricing** – restrict to p_max = 0, ensuring g(s)<1 when s<1
- **distortions** – 'ordered' shows the usual calibrated distortions, else list of dicts name:distortion.
- **title** – optional title (applied to all plots)
- **lim** – axis limits
- **check** – construct and plot Distortions to check working ; reduces n_resamples to 5

Returns

compute_weight(*premium*, *p0*, *p1*, *b=inf*, *kind='interp'*)

compute the weight for a single TVaR p0 < p1 value pair

Parameters

- **line** –
- **premium** –
- **tp** –

- **b** –

Returns

compute_weights (*line*, *premium*, *n_tps*, *b=inf*, *kind='interp'*)

Compute the weights of the extreme distortions

Applied to min(line, b) (allows to work for net)

Note: independent of the asset level

Parameters

- **line** – within port, or total
- **premium** – target premium for the line
- **n_tps** – number of tvar p points (tps)number of tvar p points (tps)number of tvar p points (tps)number of tvar p points (tps).
- **b** – loss bound: compute weights for min(line, b); generally used for net losses only.

Returns

distortion (*pl*, *pu*)

Return the BiTVaR with probabilities pl and pu

make_ps (*n*, *mode*)

If add_one then you want n = 2**m + 1 to ensure nicely spaced points.

Mode: making s points (always uniform) or tvar p points (use t_mode). self.t_mode == 'u': make uniform s points against which to evaluate g from 0 to 1 self.t_mode == 'gl': make Gauss-Legndre p points at which TVaRs are evaluated from 0 inclusive to 1 exclusive with more around 1

Parameters
n –

Returns

make_tvar_function (*line*, *b=inf*)

Change in 0.14.0 with new tvar methodology, this function reflects the b limit, it is the TVaR of min(X, b)

Make unlimited TVaR function for line, `self.tvar_unlimited_function`, and set self.Fb.

- Portfolio or Aggregate: get from object
- DataFrame: make from p_{line} column
- Series: make from Series

In the last two cases, uses aggregate.utilties.make_var_tvar_function.

Includes determining sup and putting in value for zero. If sup is largest value in index, sup set to inf.

You generally want to apply with a limit, call `self.tvar_with_bounds`.

Parameters

- **line** – only used for portfolio objects, to specify line (or 'total')
- **b** – bound on the losses, e.g., to model limited liability insurer

Returns

p_star (*line*, *premium*, *b=inf*, *kind='interp'*)

Compute p* so TVaR @ p* of min(X, b) = premium

In this case the cap b has an impact (think of integrating q(p) over p to 1, q is impacted by b)

premium <= b is required (no rip off condition)

If b < inf then must solve TVaR(p) - (1 - F(b)) / (1 - p)[TVaR(F(b)) - b] = premium Let k = (1 - F(b)) [TVaR(F(b)) - b], so solving

f(p) = TVaR(p) - k / (1 - p) - premium == 0

using NR

Parameters

- **`line`** –
- **`premium`** – target premium
- **`b`** – bound
- **`kind`** – now ignored

Returns

`ped_distortion` (*n*, *solver='rs'*)

make the approximating distortion from the first n Principal Extreme Distortions (PED)s using rs or ip solutions

Parameters
`n` –

Returns

`principal_extreme_distortion_analysis` (*gs*, *pricing=False*)

Find the principal extreme distortion analysis to solve for gs = g(s), s=self.cloud_df.index

Assumes that tvar_cloud has been called and that cloud_df exists len(gs) = len(cloud_df)

E.g., call

b = Bounds(port) b.t_mode = 'u' # set premium and asset level a b.tvar_cloud('total', premium, a) # make gs b.principal_extreme_distortion_analysis(gs)

Parameters

- **`gs`** – either g(s) evaluated on s = cloud_df.index or the name of a calibrated distortion in distribution_spec.dists (created by a call to calibrate_distortions)
- **`pricing`** – if try, try just using pricing distortions

Returns

`quick_price` (*distortion*, *a*)

price total to assets a using distortion

requires distribution_spec has a density_df dataframe with a p_total or p_total

TODO: add ability to price other lines :param distortion: :param a: :return:

`tvar_array` (*line*, *n_tps=257*, *b=inf*, *kind='interp'*)

Compute tvars at n equally spaced points, tps.

Parameters

- **`line`** –
- **`n_tps`** – number of tvar p points, default 257 (assuming add-one mode)
- **`b`** – cap on losses applied before computing TVaRs (e.g., adjust losses for finite assets b). Use np.inf for unlimited losses.
- **`kind`** – now ignored.

Returns

tvar_cloud(*line*, *premium*, *a*, *n_tps*, *s*, *kind='interp'*)

weight down tvar functions to the extremal convex measures

asset level a acts like an agg stop on what is being priced, i.e. we are working with min(X, a)

Parameters

- **line** –
- **premium** –
- **a** –
- **n_tps** –
- **s** –
- **b** – bound, applies to min(line, b)

Returns

tvar_hinges(*s*)

make the tvar hinge functions by evaluating each tvar_p(s) = min(1, s/(1-p) for p in tps, at EP points s

all arguments in [0,1] x [0,1]

Parameters

s –

Returns

tvar_with_bound(*p*, *b=inf*, *kind='interp'*)

Compute tvar taking bound into account. Assumes tvar_unfunction setup.

Warning: b must equal the b used when calibrated. The issue is computing F which varies with the type of underlying portfolio. This is fragile. Added storing b and checking equal. For backwards comp. need to keep b argument

Parameters

- **p** –
- **b** –
- **kind** – now ignored

Returns

aggregate.bounds.**plot_lee**(*port*, *ax*, *c*, *lw=1*)

Lee diagram by hand

aggregate.bounds.**plot_max_min**(*self*, *ax*)

Extracted from bounds, self=Bounds object

aggregate.bounds.**similar_risks_example**()

Interesting beta risks and how to use similar_risks_graphs_sa.

Returns

aggregate.bounds.**similar_risks_graphs_sa**(*axd*, *bounds*, *port*, *pnew*, *roe*, *prem*, *p_reg=1*)

stand-alone ONLY WORKS FOR BOUNDED PORTFOLIOS (use for beta mixture examples) Updated version in CaseStudy axd from mosaic bounds = Bounds class from port (calibrated to some base)it pnew = new portfolio input new beta(a,b) portfolio, using existing bounds object

sample: see similar_risks_sample()

Provenance : from make_port in Examples_2022_post_publish

3.8 Extensions

Extensions contains optional, nice to have code that extends basic functionality. For example, all CaseStudy material is included here. It is not required for core functionality. It is not included in the default import. It is included in the default build.

3.8.1 Basic

3.8.2 Case Study Support

class `aggregate.extensions.case_studies.`**`ClassicalPremium`**(*ports*, *calibration_premium*)

manage classical premium examples

Net, no loading Expected value, constant loading Maximum loss –> no VaR (as proxy for maximum loss) Variance Std Dev Semi-variance (Artzner p. 210) Exponential (zero utility, convex!) Esscher

Originally in hack.py

`calibrate`(*port_name*, *line_name*, *calibration_premium*, *df=None*, *ob=None*, *stats=None*, *mn=None*, *var=None*, *sd=None*)

calibrate all methods…

`distribution`(*port_name*, *line_name*)

classical methods all depend on the distribution…so pull it out pull the object that will provide q etc. pull the audit stats

`illustrate`(*port_name*, *line_name*, *ax*, *margin*, ***, *p=0*, *K=0*, *n_big=10000*, *n_sample=25*, *show_bounds=True*, *padding=2*)

illustrate simulations at p level probability probability level determines capital or capital K input margin: premium = (1 + margin) * EL, margin = rho n_big = number of policies - max of horizontal axis n_sample = number of iterations to plot

Theoretic bounds use the actual moments, simulated use those from the process being estimated

From common_scripts Pentagon took out re-computation…

`price`(*param*, *port_name*, *line_name*, *method*, *df=None*, *ob=None*, *stats=None*, *mn=None*, *var=None*, *sd=None*)

apply method to port_name, line_name with parameter(s) (all one param) these are all classical methods

method_dict = {method name : param } param = float | [float param, p ge 1 value] latter fro Fischer method

`prices`(*port_name*, *line_name*, *method_dict*)

run lots of prices

`pricing_exhibit`(*port_name*, *line_name*, *calibration_premium*, *re_line*)

calibrate and apply to all other portfolios

`aggregate.extensions.case_studies.`**`add_defaults`**(*dict_in*, *kind='agg'*)

add default values to dict_inin. Leave existing values unchanged Used to output to a data frame, where you want all columns completed

Parameters

- **`dict_in`** –
- **`kind`** –

Returns

aggregate.extensions.case_studies.**bivariate_density_plots**(*axi*, *ports*, *xmax*, *contour_scale*, *biv_log=True*, *cmap='viridis'*, *levels=30*, *color_bar=False*)

bivarate plots of each line against the others for each portfolio in case arguments as for twelve_plot / bivden plot

axi = iterator with enough exes ports = iterable of ports (list, dict.values(), etc.)

from common_scripts.py

aggregate.extensions.case_studies.**extract_sort_order**(*summaries*, *_varlist_*, *classical=False*)

Pull out exhibits. Note difference: classical uses net_classical, calibrated to roe non-classical uses pricing calibrated to

aggregate.extensions.case_studies.**g_ins_stats**(*axi*, *dn0*, *dist*, *ls='-'*)

Six part plot with EL, premium, loss ratio, profit, layer ROE and P:S for g axi = axis iterator

aggregate.extensions.case_studies.**macro_market_graphs**(*axi*, *port*, *dn*, *rp*)

Create more succinct 4x4 graph to illustrate macro market structure and result, LR, P:S and ROE, etc.

Use a port object with calibrated distortion

see ch04_macro_market_stats_original

from: common_scripts.py ch04_macro_market_stats updated line colors

June 2022: removed all color='k'

Parameters

- **dist** –
- **rp** – return period
- **sigma** –

Returns

aggregate.extensions.case_studies.**pricing**(*port*, *p*, *roe*, *as_dataframe=True*)

Make nice stats output for pricing from common_scripts.py

aggregate.extensions.case_studies.**universal_renamer**(*x*)

aggregate.extensions.case_studies.**urn**(*df*)

apply universal renamer

3.8.3 Pentagon

aggregate.extensions.pentagon.**code**(*r*)

determine binary code for row

aggregate.extensions.pentagon.**make_possible_pentagons**()

enumerate possible and impossible pentagon configurations

class aggregate.extensions.pentagon.**pent_ans**(*L*, *P*, *M*, *a*, *Q*, *LR*, *PQ*, *COC*)

COC

Alias for field number 7

L

Alias for field number 0

LR

Alias for field number 5

M

Alias for field number 2

P

Alias for field number 1

PQ

Alias for field number 6

Q

Alias for field number 4

a

Alias for field number 3

3.8.4 Samples

Warning: All functionality in `extensions.samples` has been moved into the base package.

3.8.5 Figures

`aggregate.extensions.figures.`**`adjusting_layer_losses`**`()`

Figure to illustrate the process of adjusting layer losses. TODO: Add reference

`aggregate.extensions.figures.`**`discretization_agg_example`**`(`*outcomes*`)`

For AAS paper. Convergence of sevs with smaller bucket size.

`aggregate.extensions.figures.`**`discretization_sev_example`**`(`*outcomes*`)`

For AAS paper. Convergence of sevs with smaller bucket size.

`aggregate.extensions.figures.`**`dual_distortion`**`(`*dist=None*, *s=0.3*`)`

Illustrate how the dual distortion relates to the distortion.

`aggregate.extensions.figures.`**`g_insurance_statistics`**`(`*axi*, *dist*, *c='C0'*, *ls='-'*, *lw=1*, *diag=False*, *grid=False*`)`

Six part plot with EL, premium, loss ratio, profit, layer ROE and P:S for g axi = axis iterator with 6 axes dist = distortion Used to create PIR Figs 11.6 and 11.7

`aggregate.extensions.figures.`**`g_risk_appetite`**`(`*axi*, *dist*, *c='C0'*, *ls='-'*, *N=1000*, *lw=1*, *grid=False*, *xlabel=True*, *title=True*, *add_tvar=False*`)`

Plot to illustrate the risk appetite associated with a distortion g. Derived from `g_insurance_statistics`. Plots premium, loss ratio, margin, return (easier to understand than discount), VaR wts and optionally TVaR wts axi = axis iterator with 6 axes dist = distortion Used to create PIR Figs 11.6 and 11.7

`aggregate.extensions.figures.`**`gh_example`**`(`*en*`)`

Code to reproduce GHGrübel and Hermesmeier 1999, Table 1. The function `exact_cdf` calculates the compound probability that $x - 1/2 < X \le x + 1/2$. For AAS paper with en=20.

`aggregate.extensions.figures.`**`mixing_convergence`**`(`*freq_cv*, *sev_cv*, *bs=0.015625*`)`

Illustrate convergence of mixed distributions to the mixing distribution.

aggregate.extensions.figures.**power_variance_family**()

Graph to illustrate the power variance exponential family distributions.

Reference: Jørgensen, Bent. 1997. The theory of dispersion models. CRC Press.

aggregate.extensions.figures.**savings_charge**()

Figure to illustrate the insurance savings and expense(charge).

3.8.6 PIR Figures

aggregate.extensions.pir_figures.**curlyBrace**(*ax*, *p1*, *p2*, *k_r=0.1*, *bool_auto=True*, *str_text=''*, *int_line_num=2*, *fontdict={}*, ***kwargs*)

Plot an optionally annotated curly bracket on the given axes of the given figure.

Note that the brackets are anti-clockwise by default. To reverse the text position, swap "p1" and "p2".

Note that, when the axes aspect is not set to "equal", the axes coordinates need to be transformed to screen coordinates, otherwise the arcs may not be seeable.

Parameters

fig
[matplotlib figure object] The of the target axes.

ax
[matplotlib axes object] The target axes.

p1
[two element numeric list] The coordinates of the starting point.

p2
[two element numeric list] The coordinates of the end point.

k_r
[float] This is the gain controlling how "curvy" and "pointy" (height) the bracket is.

Note that, if this gain is too big, the bracket would be very strange.

bool_auto
[boolean] This is a switch controlling wether to use the auto calculation of axes scales.

When the two axes do not have the same aspects, i.e., not "equal" scales, this should be turned on, i.e., True.

When "equal" aspect is used, this should be turned off, i.e., False.

If you do not set this to False when setting the axes aspect to "equal", the bracket will be in funny shape.

Default = True

str_text
[string] The annotation text of the bracket. It would displayed at the mid point of bracket with the same rotation as the bracket.

By default, it follows the anti-clockwise convention. To flip it, swap the end point and the starting point.

The appearance of this string can be set by using "fontdict", which follows the same syntax as the normal matplotlib syntax for font dictionary.

Default = empty string (no annotation)

int_line_num
[int] This argument determines how many lines the string annotation is from the summit of the bracket.

The distance would be affected by the font size, since it basically just a number of lines appended to the given string.

Default = 2

fontdict
[dictionary] This is font dictionary setting the string annotation. It is the same as normal matplotlib font dictionary.

Default = empty dict

****kwargs**
[matplotlib line setting arguments] This allows the user to set the line arguments using named arguments that are the same as in matplotlib.

Returns

theta
[float] The bracket angle in radians.

summit
[list] The positions of the bracket summit.

arc1
[list of lists] arc1 positions.

arc2
[list of lists] arc2 positions.

arc3
[list of lists] arc3 positions.

arc4
[list of lists] arc4 positions.

Reference

https://uk.mathworks.com/matlabcentral/fileexchange/38716-curly-brace-annotation

`aggregate.extensions.pir_figures.`**`fig_10_3`**`(`*dist=None*, *s=0.3*`)`

Figure 10.3 Illustrating distortion functions (s, g(s)) with vertical line at s and split loss, premium, margin, and capital labelled

`aggregate.extensions.pir_figures.`**`fig_10_5`**`(`*port=None*, *dist=None*, *s=0.3*`)`

three plot version of previous with more explanation of first picture

return_period_max = defines extend of yaxis return_period_x = capital level to illustrate

map from s space into loss space extended version of ch04_s_gs_loss_premium_capital which includes the horizontal bar [loss][m][equity] plotted on the provided second axis

Suggested figure set up for extended:

f = plt.Figure(figsize=(4,3), tight_layout=True) a = f.add_axes([0, 100/3+1/27, 1, 2/3], label='a') b = f.add_axes([0, 0, 1, 1/3], label='b')

`aggregate.extensions.pir_figures.`**`fig_10_6`**`(`*port=None*, *dist=None*`)`

Same distortion and portfolio as 10_5 Slight clarification of the diagram vs. book version.

`aggregate.extensions.pir_figures.`**`fig_4_1`**`()`

Figure 4.1: illustrating quantiles.

`aggregate.extensions.pir_figures.`**`getAxSize`**`(`*fig*, *ax*`)`

Get the axes size in pixels.

Reference: https://uk.mathworks.com/matlabcentral/fileexchange/38716-curly-brace-annotation

Parameters

- **`fig`** – matplotlib figure object The of the target axes.
- **`ax`** – matplotlib axes object The target axes.

Returns
ax_width : float, the axes width in pixels; ax_height : float, the axes height in pixels.

`aggregate.extensions.pir_figures.`**`natural_scale`**`(`*port*`)`

For creating Table 9.15

3.8.7 Test Suite

CHAPTER

FOUR

DEC LANGUAGE REFERENCE

This section describes how a DecL program is pre-processed, lexed, and parsed according to the grammar specification. It reports the results of interpreting the builtin test suite of programs.

The DecL *introduction* describes its design and purpose.

4.1 Pre-Processing

Programs are processed one line at a time. Before passing to the lexer, the following pre-processing occurs.

1. Remove Python and C++ style `#` or `//` comments, through end of line
2. Remove \n in [ ] (vectors) that appear from using `f'{np.linspace(...)}'`
3. Map semicolons to newline
4. Map backslash newline (Python line continuations) to space
5. Replace \n\t with space, to support the tabbed indented Portfolio layout
6. Split on remaining newlines

4.2 Lexer Term Definitions

Ignored characters: tab (remaining after pre-processing), colon, comma, and pipe. These characters can be used to improve readability.

Aggregate names must not include underscore. Portfolio names may include underscore. Names can include a period, `A.Basic.01`.

```
tokens = {ID, BUILTIN_AGG, BUILTIN_SEV,NOTE,
          SEV, AGG, PORT,
          NUMBER, INFINITY,
          PLUS, MINUS, TIMES, DIVIDE, INHOMOG_MULTIPLY,
          LOSS, PREMIUM, AT, LR, CLAIMS, EXPOSURE, RATE,
          XS, PICKS,
          DISTORTION,
          CV, WEIGHTS, EQUAL_WEIGHT, XPS,
          MIXED, FREQ, TWEEDIE, ZM, ZT,
          NET, OF, CEDED, TO, OCCURRENCE, AGGREGATE, PART_OF, SHARE_OF, TOWER,
          AND,  PERCENT,
          EXPONENT, EXP,
          DFREQ, DSEV, RANGE
          }

ignore = ' \t,\\|'
literals = {'[', ']', '!', '(', ')'}
```

(continues on next page)

(continued from previous page)

```
NOTE = r'note\{[^\}]*\}'  # r'[^\}]+'
BUILTIN_AGG = r'agg\.[a-zA-Z][a-zA-Z0-9._:~]*'
BUILTIN_SEV = r'sev\.[a-zA-Z][a-zA-Z0-9._:~]*'
FREQ = 'binomial|pascal|poisson|bernoulli|geometric|fixed|neyman(a|A)?|logarithmic'
DISTORTION = 'dist(ortion)?'
NUMBER = r'\-?(\d+\.?\d*|\d*\.\d+)([eE](\+|\-)?\d+)?'
ID = r'[a-zA-Z][\.:~_a-zA-Z0-9]*'
EXPONENT = r'\^|\*\*'
PLUS = r'\+'
MINUS = r'\-'
TIMES = r'\*'
DIVIDE = '/'
PERCENT = '%'
INHOMOG_MULTIPLY = '@'
EQUAL_WEIGHT = '='
RANGE = ':'

ID['occurrence'] = OCCURRENCE
ID['unlimited'] = INFINITY
ID['aggregate'] = AGGREGATE
ID['exposure'] = EXPOSURE
ID['tweedie'] = TWEEDIE
ID['premium'] = PREMIUM
ID['tower'] = TOWER
ID['mixed'] = MIXED
ID['unlim'] = INFINITY
ID['picks'] = PICKS
ID['prem'] = PREMIUM
ID['claims'] = CLAIMS
ID['ceded'] = CEDED
ID['claim'] = CLAIMS
ID['dfreq'] = DFREQ
ID['dsev'] = DSEV
ID['loss'] = LOSS
ID['port'] = PORT
ID['rate'] = RATE
ID['net'] = NET
ID['sev'] = SEV
ID['agg'] = AGG
ID['xps'] = XPS
ID['wts'] = WEIGHTS
ID['inf'] = INFINITY
ID['and'] = AND
ID['exp'] = EXP
ID['wt'] = WEIGHTS
ID['at'] = AT
ID['cv'] = CV
ID['lr'] = LR
ID['xs'] = XS
ID['of'] = OF
ID['to'] = TO
ID['po'] = PART_OF
ID['so'] = SHARE_OF
ID['zm'] = ZM
ID['zt'] = ZT
ID['x'] = XS
```

4.3 Dec Language Grammar Specification

Here is the full DecL Grammar and a grammar railroad diagram.

```
answer                      ::= agg_out
                             | port_out
                             | distortion_out
                             | expr

distortion_out              ::= DISTORTION name ID expr
                             | DISTORTION name ID expr "[" numberl "]"

port_out                    ::= PORT name note agg_list

agg_list                    ::= agg_list agg_out
                             | agg_out

agg_out                     ::= AGG name exposures layers sev_clause occ_reins␣
↪freq agg_reins note
                             | AGG name dfreq layers sev_clause occ_reins agg_
↪reins note
                             | AGG name TWEEDIE expr expr expr note
                             | AGG name builtin_agg occ_reins agg_reins note
                             | builtin_agg agg_reins note

sev_out                     ::= SEV name sev note
                             | SEV name dsev note

freq                        ::= freq ZM expr
                             | freq ZT
                             | MIXED ID expr expr
                             | MIXED ID expr
                             | FREQ expr expr
                             | FREQ expr
                             | FREQ

agg_reins                   ::= AGGREGATE NET OF reins_list
                             | AGGREGATE CEDED TO reins_list
                             |  %prec LOW

occ_reins                   ::= OCCURRENCE NET OF reins_list
                             | OCCURRENCE CEDED TO reins_list
                             |

reins_list                  ::= reins_list AND reins_clause
                             | reins_clause
                             | tower

reins_clause                ::= expr XS expr
                             | expr SHARE_OF expr XS expr
                             | expr PART_OF expr XS expr

sev_clause                  ::= SEV sev
                             | dsev
                             | BUILTIN_SEV

sev                         ::= sev picks
                             | sev "!"
                             | sev2 weights splice
                             | BUILTIN_SEV

dsev                        ::= dsev "!"
```

(continues on next page)

(continued from previous page)

```
                            | DSEV doutcomes dprobs

sev2                       ::= sev1 PLUS numbers
                            | sev1 MINUS numbers
                            | sev1

sev1                       ::= numbers TIMES sev0
                            | sev0

sev0                       ::= ids numbers CV numbers
                            | ids numbers numbers
                            | ids numbers
                            | ids xps
                            | ids

xps                        ::= XPS doutcomes dprobs

dfreq                      ::= DFREQ doutcomes dprobs

picks                      ::= PICKS "[" numberl "]" "[" numberl "]"

doutcomes                  ::= "[" numberl "]"
                            | "[" expr RANGE expr "]"
                            | "[" expr RANGE expr RANGE expr "]"

dprobs                     ::= "[" numberl "]"
                            |

weights                    ::= WEIGHTS EQUAL_WEIGHT expr
                            | WEIGHTS "[" numberl "]"
                            |

splice                     ::= SPLICE "[" numberl "]"
                            |

layers                     ::= numbers XS numbers
                            | tower
                            |

tower                      ::= TOWER doutcomes

note                       ::= NOTE
                            |  %prec LOW

exposures                  ::= numbers CLAIMS
                            | numbers LOSS
                            | numbers PREMIUM AT numbers LR
                            | numbers EXPOSURE AT numbers RATE

ids                        ::= "[" idl "]"
                            | ID

idl                        ::= idl ID
                            | ID

builtin_agg                ::= expr INHOMOG_MULTIPLY builtin_agg
                            | expr TIMES builtin_agg
                            | builtin_agg PLUS expr
                            | builtin_agg MINUS expr
                            | BUILTIN_AGG
```

(continues on next page)

(continued from previous page)

```
name                            ::= ID

numbers                         ::= "[" numberl "]"
                                  | "[" expr RANGE expr "]"
                                  | "[" expr RANGE expr RANGE expr "]"
                                  | expr

numberl                         ::= numberl expr
                                  | expr

expr                            ::= atom

atom                            ::= atom DIVIDE atom
                                  | "(" atom ")"
                                  | EXP atom
                                  | atom EXPONENT atom
                                  | NUMBER

FREQ                        ::= 'binomial|poisson|bernoulli|pascal|geometric|neymana?
↪|fixed|logarithmic|negbin'

BUILTINID                   ::= 'sev|agg|port|meta.ID'

NOTE                        ::= 'note{TEXT}'

EQUAL_WEIGHT                ::= "="

AGG                         ::= 'agg'

AGGREGATE                   ::= 'aggregate'

AND                         ::= 'and'

AT                          ::= 'at'

CEDED                       ::= 'ceded'

CLAIMS                      ::= 'claims|claim'

CONSTANT                    ::= 'constant'

CV                          ::= 'cv'

DFREQ                       ::= 'dfreq'

DSEV                        ::= 'dsev'

EXP                         ::= 'exp'

EXPONENT                    ::= '^|**'

INHOMOG_MULTIPLY            ::= "@"

INFINITY                    ::= 'inf|unlim|unlimited'

LOSS                        ::= 'loss'

LR                          ::= 'lr'

MIXED                       ::= 'mixed'
```

(continues on next page)

(continued from previous page)

```
NET                         ::= 'net'

OCCURRENCE                  ::= 'occurrence'

OF                          ::= 'of'

PART_OF                     ::= 'po'

PERCENT                     ::= '%'

PORT                        ::= 'port'

PREMIUM                     ::= 'premium|prem'

SEV                         ::= 'sev'

SHARE_OF                    ::= 'so'

TO                          ::= 'to'

WEIGHTS                     ::= 'wts|wt'

XPS                         ::= 'xps'

XS                          ::= "xs|x"
```

4.4 Test Suite Programs

To run the test suite for HTML output, svg graphics.

```
from aggregate.extensions.test_suite import TestSuite
TestSuite().run('^[A-KNO]', 'All Aggregate Tests', 'svg')
```

```
# Test Suite Examples
# ===================

# Comprehensive list of examples to test the parser and creation logic.

# Contents
#
# A. Creating Aggregates, Portfolios, and Distortion objects
# B. Basic examples using dfreq and dsev notation, including dice examples
# C. Frequency only fixed sev
# D. Severity only, fixed freq, different distributions
# E. Severity transformations: shift, scale, layer, and attachment, unconditional␣
↪severities
# F. Specifying exposure
# G. Mixed severities
# H. Limit profiles
# I. Limit profiles with mixed severities
# J. Reinsurance
# K. Tweedie distribution examples
# L. Examples from papers
# M. Case Studies from Pricing Insurance Risk book
# N. Proxies for US Lines
# O. Novelties (the logo mixed distribution)
```

(continues on next page)

(continued from previous page)

```
# Default severity used in many other examples
sev One dsev [1]

# Guides examples
# ===============

# Simple Dice Examples
# ======================
agg A.Dice00 dfreq [1:6] dsev [1]                   note{The roll of a single dice.
↪}
agg A.Dice01 dfreq [1]   dsev [1:6]                 note{Same as previous example.}
agg A.Dice02 dfreq [2]   dsev [1:6]                 note{Sum of the rolls of two␣
↪dice.}
agg A.Dice03 dfreq [5]   dsev [1:6]                 note{Sum of the rolls of five␣
↪dice.}
agg A.Dice04 dfreq [1:6] dsev [1:6]                 note{Sum of a dice roll of␣
↪dice rolls}
agg A.Dice05 dfreq [1:4] dsev [1:16]                note{Something you can't do␣
↪easily by hand}

# Basic examples using freq and dsev notation
# ===========================================
agg B.Basic01 dfreq [1] dsev [0 1]                            note{toss of a␣
↪single coin}
agg B.Basic02 dfreq [12] dsev [0 1]                           note{toss of␣
↪12 single coins}
agg B.Basic03 dfreq [1:3] dsev [1 2 10]                       note{1, 2 or 3␣
↪claims using range notation, sev 1, 2, 10 all equally likely}
agg B.Basic03 dfreq [1:11:2] dsev [1 2 10]                    note{1 3...11␣
↪claims range and step, sev 1, 2, 10 all equally likely}
agg B.Basic04 dfreq [1 2 3] [.5 1/4 1/4] dsev [1 2 10]        note{specify␣
↪probabilities of claims}
agg B.Basic05 dfreq [1:3]   dsev [1 2 10] [.4 .4 .2]          note{specify␣
↪probabilities of sev}
agg B.Basic06 dfreq [1:3] [.5 1/4 1/4] dsev [1 2 10] [.4 .4 .2]     note{specify␣
↪both probabilities }
agg B.Basic07 dfreq [0 1 2] [.5 .3 .2] sev sev.One
agg B.Basic08 dfreq [0 1 2] [.5 .3 .2] sev.One
agg B.Basic09 dfreq [0 1 2] [.5 .3 .2] sev lognorm 10 cv .3

# Frequency only, fixed severity
# ==============================
# Using the usual exposure-based frequency clause
agg Ca.Freq01.Fixed         10 claims sev.One fixed
agg Ca.Freq02.Poisson       10 claims sev.One poisson
agg Ca.Freq03.Bernoulli     .8 claims sev.One bernoulli
agg Ca.Freq04.Binomial      10 claims sev.One binomial 0.5
agg Ca.Freq05.Geometric     10 claims sev.One geometric
agg Ca.Freq06.Pascal        10 claims sev.One pascal .8 3
agg Ca.Freq07.NegBin        10 claims sev.One negbin 3             note{shape␣
↪paramter equals variance multiplier}
agg Ca.Freq08.Logarithmic   10 claims sev.One logarithmic
agg Ca.Freq09.NeymanA       10 claims sev.One neymana 3            note{shape␣
↪paramter number of eggs per cluster}
# Mixing distributions
agg Cb.Freq10.NegBin        10 claims sev.One mixed gamma 0.65     note{shape␣
↪paramter equals variance of mixing distribution}
agg Cb.Freq11.Delaporte     10 claims sev.One mixed delaporte .65 .25
agg Cb.Freq12.IG            10 claims sev.One mixed ig .65
agg Cb.Freq13.SIG           10 claims sev.One mixed sig 0.5 0.4
```

(continues on next page)

(continued from previous page)

```
agg Cb.Freq14.Sichel       10 claims sev.One mixed delaporte .65 -0.25
agg Cb.Freq15.Sichel.gamma 10 claims sev.One mixed sichel.gamma .65 .25
agg Cb.Freq16.Sichel.ig    10 claims sev.One mixed sichel.ig .65 .25
agg Cb.Freq17.Beta         10 claims sev.One mixed beta .5 4
# ZM and ZT distributions, note alternative way to specify fixed severity
agg Cc.Freq20.Poisson     4 claims dsev [1] poisson zt
agg Cc.Freq21.Poisson     4 claims dsev [1] poisson zm .5
agg Cc.Freq22.Geometric   4 claims dsev [1] geometric zm .5        note{pr(N=0)=1,␣
↪there is no zt version}
agg Cc.Freq23.Logarithmic 4 claims dsev [1] logarithmic zm .5      note{pr(N=0)=1,␣
↪there is no zt version}
agg Cc.Freq24.Binomial    4 claims dsev [1] binomial 0.6 zm .5     note{Cannot␣
↪make p0 smaller than the natural p}
agg Cc.Freq25.Negbin      4 claims dsev [1] negbin 3 zm .5         note{Limits on␣
↪p0}

# Severity only, fixed freq, different distributions
# ==================================================
# zero parameter
agg D.Sev01 1 claim sev 100 * expon        fixed                    note{zero␣
↪param severity can look odd, but it works}
agg D.Sev02 1 claim sev 100 * expon 1 + 10 fixed
agg D.Sev03 1 claim sev 100 * expon   + 10 fixed                    note{will this␣
↪work?}
agg D.Sev04 1 claim sev 100 * norm    +500 fixed
agg D.Sev05 1 claim sev 100 * uniform + 50 fixed
agg D.Sev05b 2 claims sev 100*uniform + 50 fixed
# one parameter
agg D.Sev06 1 claim sev 10 * gamma 0.3      fixed
agg D.Sev07 1 claim sev      gamma 12 cv .3 fixed
agg D.Sev08 1 claim sev      lognorm 50 cv .3 fixed
agg D.Sev09 1 claim sev 50/exp(.3**2/2) * lognorm .3 fixed          note{mean␣
↪equals 50 and cv is slightly higher than .3}
agg D.Sev10 1 claim sev 100 * invgamma 4.07 fixed                   note{remember␣
↪must set scale or mean}
agg D.Sev11 1 claim sev 100 * weibull_min 1.5 fixed
agg D.Sev12 1 claim sev invgauss 10 cv .5 fixed
agg D.Sev13 1 claim sev 10 * pareto 2.6   + -10 fixed               note{entering␣
↪Pareto is awkward}
# two parameter distributions
agg D.Sev14 1 claim sev 50 * beta    3 2 + 10 fixed
# empirical severities, continuous and discrete
agg D.Sev15 1   claim  sev  dhistogram xps [1 10 40] [.5 .3 .2] fixed  note{old␣
↪notation}
agg D.Sev16 dfreq [1] dsev [1 10 40] [.5 .3 .2]                        note
↪{preferred new notation}
agg D.Sev17 1   claim  sev chistogram xps [1 10 40] [.5 .3 .2] fixed   note
↪{continuous version}

# Severity transformations: shift, scale, layer, and attachment, unconditional␣
↪severities
#␣
↪==========================================================================================
agg E.TSev00  1 claim sev      lognorm 10 cv .09 fixed
agg E.TSev01  1 claim sev 10 * lognorm 10 cv .09 fixed
agg E.TSev02  1 claim sev      lognorm 10 cv .09 + 20 fixed
agg E.TSev03  1 claim sev 10 * lognorm 10 cv .09 + 20 fixed
# here, enter shape parameter directly rather than use mean and CV
agg E.TSev04  1 claim sev 9.559974818331 * lognorm .3 fixed
agg E.TSev05  1 claim sev 9.559974818331 * lognorm .3 + 5 fixed
# with layer and attachments
```

(continues on next page)

(continued from previous page)

```
agg E.TSev06  1 claim 130 xs 20 sev      lognorm 20 cv 0.75     fixed
agg E.TSev07  1 claim 130 xs 20 sev 20 * lognorm 0.75           fixed
agg E.TSev08  1 claim 130 xs 20 sev 20 * lognorm 1 cv 0.75      fixed
agg E.TSev09  1 claim 130 xs 20 sev 20 * lognorm 1 cv 0.75 + 20 fixed
# ground up and unlimited
agg E.TSev10 10 claims           sev lognorm 10 cv 0.8 poisson
agg E.TSev11 10 claims  30 xs  0 sev lognorm 10 cv 0.8 poisson
agg E.TSev12 10 claims 100 xs 10 sev lognorm 10 cv 0.8 poisson
agg E.TSev13 10 claims inf xs  0 sev lognorm 10 cv 0.8 poisson
agg E.TSev14 10 claims inf xs 10 sev lognorm 10 cv 0.8 poisson
# unconditional severity
agg E.TSev15 5 claims 200 xs  0 sev 1.2343e2 * lognorm 2    poisson          note
↪{conditional severity}
agg E.TSev16 5 claims 200 xs  0 sev 1.2343e2 * lognorm 2  ! poisson          note
↪{unconditional severity}
agg E.TSev17 5 claims 200 xs 10 sev 1.2343e2 * lognorm 2    poisson          note
↪{conditional severity}
agg E.TSev18 5 claims 200 xs 10 sev 1.2343e2 * lognorm 2  ! poisson          note
↪{unconditional severity}

# Specifying exposure
# ===================
agg F.Expos01   10 claims         sev lognorm 50 cv 0.8 poisson    note{specify␣
↪number of claims}
agg F.Expos02  500 loss           sev lognorm 50 cv 0.8 poisson    note{specify␣
↪expected loss, derive number of claims}
agg F.Expos03 1000 prem at .5 lr  sev lognorm 50 cv 0.8 poisson    note{specify␣
↪premium and loss ratio, derive number of claims}

# Mixed and Spliced severities
# ============================
agg G.Mixed00  1  claim  50 xs 0 sev lognorm 10 cv [0.2 0.4 0.6 0.8 1.0] wts [.2 .
↪3 .3 .15 .05]              poisson            note{no shared mixing}
agg G.Mixed01  1  claim  50 xs 0 sev lognorm 10 cv [0.2 0.4 0.6 0.8 1.0] wts [.2 .
↪3 .3 .15 .05]              mixed gamma 0.3    note{shared mixing, compare audit␣
↪and report dfs}
agg G.Mixed02  1  claim  50 xs 0 sev lognorm 10 cv [0.2 0.4 0.6 0.8 1.0] wts=5    ␣
↪                          mixed gamma 0.3
agg G.Mixed03  1  claim  50 xs 0 sev lognorm 10 cv [0.2 0.4 0.6 0.8 1.0]          ␣
↪                          mixed gamma 0.3
agg G.Mixed04  1  claim  50 xs 0 sev lognorm [2 4 6 8 10] cv 1           wts [.2 .
↪3 .3 .15 .05]              mixed gamma 0.3
agg G.Mixed05  1  claim 250 xs 0 sev lognorm [10 15 20 25 50 100] cv [0.1 0.2 0.4␣
↪0.6 0.8 1.0] wts=6         mixed gamma 0.3
agg G.Mixed07  1  claim          sev 100                *  beta [1 200 500 100]␣
↪[100 800 500 1] + 10   wts=4  mixed gamma 0.3
agg G.Mixed08  1  claim          sev [100 200 250 300] *  beta [1 200 500 100]␣
↪[100 800 500 1] + 10   wts=4  mixed gamma 0.3
agg G.Mixed09  8  claim          sev     100 * [lognorm expon] [.5 1] wts [0.6 .
↪4]                          mixed gamma 0.3    note{different severities}
agg G.Mixed10  1  claim          sev [50 100] * [lognorm expon]  [2 1]   + 10 wts=2␣
↪                          mixed gamma 0.3
agg G.Mixed11  1  claim          sev [50 100] * [lognorm expon]  [2 1]   + 10      ␣
↪                          mixed gamma 0.3

# Limit profiles
# ==============
agg H.Limits01               1 claim [1 5 10 20] xs  0              sev lognorm 10␣
↪cv 1.2  fixed
agg H.Limits02               5 claim            100 xs [0 10 50]    sev lognorm 10␣
↪cv 1.2  fixed
```

(continues on next page)

(continued from previous page)

```
agg H.Limits02                5 claim [10 20 50 100] xs [0 0 50 100] sev lognorm 10␣
↪cv 2.0  fixed
agg H.Limits03   [10 10 10 10] claims [inf 10 inf 10] xs [0 0 5 5]  sev lognorm 10␣
↪cv 1.25 fixed

# Limit profiles with mixed severities
# ====================================
agg I.Blend01  10 claims [5 10 15] xs 0                sev lognorm 12 cv [1 1.5␣
↪3]                 poisson
agg I.Blend02  10 claims [5 10 15] xs 0                sev lognorm 12 cv [1 1.5␣
↪3]                 mixed gamma 0.25
agg I.Blend03  10 claims [5 10 15] xs 0                sev lognorm 12 cv [1 1.5␣
↪3] wts=3           mixed gamma 0.25
agg I.Blend04   1 claims [1 5 10 20] xs  0         sev lognorm 10 cv 1.2 wts [.50␣
↪.20 .20 .1]      mixed gamma 0.25
agg I.Blend05   5 claims [10 20 50 100] xs 10         sev lognorm 10 cv 1.2 wts [.
↪50 .20 .20 .1]  mixed gamma 0.25
agg I.Blend06 [10 30 15 5]  claims [inf 10 inf 10] xs [0 0 5 5] sev lognorm  10 cv␣
↪[1.0 1.25 1.5] wts=3         mixed gamma 0.25
agg I.Blend07   [10 20 30]  claims [100 200 75] xs [0 50 75]     sev lognorm 100 cv␣
↪[1 2] wts [.6 .4]            mixed gamma 0.4
agg I.Blend08      [10 30]  claims                              sev lognorm 100 cv␣
↪[1 2]                        mixed gamma 0.4
agg I.Blend09  [1000 2000 500] prem at [.8 .7 .5] lr            sev lognorm 10 cv␣
↪[.2 .35 .5] wts [1/2 3/8 1/8] mixed gamma 0.5  note{log2=17;}
agg I.Blend10  [500 800 200]   loss                             sev lognorm 10 cv␣
↪[.2 .35 .5] wts=3             mixed gamma 0.5
agg I.Blend11  [1000 2000 500] prem at [.8 .7 .5] lr            sev lognorm 10 cv␣
↪[.2 .35 .5] wts [1/2 3/8 1/8] mixed gamma 0.5  note{log2=17;}
agg I.Blend12  [500 800 200]   loss                             sev lognorm 10 cv␣
↪[.2 .35 .5] wts=3             mixed gamma 0.5

# Reinsurance
# ===========
agg J.Re01  5 claims 100 xs 0 sev lognorm 10 cv .75 occurrence net of 50% so 5 xs␣
↪0 and 5 po 15 xs 5 and 30 xs 20   poisson
agg J.Re02  5 claims 100 xs 0 sev lognorm 10 cv .75                              ␣
↪                                  poisson aggregate net of 50% so 25 xs 0 and 75␣
↪xs 25
agg J.Re03  5 claims 100 xs 0 sev lognorm 10 cv .75 occurrence net of 50% so 5 xs␣
↪0 and 5 po 15 xs 5 and 30 xs 20   poisson aggregate net of 50% so 25 xs 0 and␣
↪100 xs 25
agg J.Re04  5 claims 100 xs 0 dsev [1:100]          occurrence net of 50% so 5 xs␣
↪0 and 5 po 15 xs 5 and 30 xs 20   poisson aggregate net of 50% so 25 xs 0 and␣
↪100 xs 25
agg J.Re05  5 claims 100 xs 0 sev lognorm 10 cv .75 occurrence ceded to 50% so 5␣
↪xs 0 and 5 po 15 xs 5 and 30 xs 20 poisson
agg J.Re06  5 claims 100 xs 0 sev lognorm 10 cv .75                              ␣
↪                                  poisson aggregate ceded to 50% so 25 xs 0 and␣
↪75 xs 25
agg J.Re07  5 claims 100 xs 0 sev lognorm 10 cv .75 occurrence ceded to 50% so 5␣
↪xs 0 and 5 po 15 xs 5 and 30 xs 20 poisson aggregate ceded to 50% so 25 xs 0 and␣
↪100 xs 25
agg J.Re08  5 claims 100 xs 0 dsev [1:100]          occurrence ceded to 50% so 5␣
↪xs 0 and 5 po 15 xs 5 and 30 xs 20 poisson aggregate ceded to 50% so 25 xs 0 and␣
↪100 xs 25
agg J.Re09  5 claims 100 xs 0 sev lognorm 10 cv .75 occurrence ceded to 15 xs 5␣
↪poisson
agg J.Re10  5 claims 100 xs 0 sev lognorm 10 cv .75                            ␣
↪poisson aggregate net of  20 xs 0
agg J.Re11  5 claims 100 xs 0 sev lognorm 10 cv .75 occurrence ceded to 15 xs 5␣
```

(continues on next page)

(continued from previous page)

```
↪poisson aggregate net of   20 xs 0
agg J.Re12  5 claims 100 xs 0 dsev [1:100]          occurrence ceded to 15 xs 5␣
↪poisson aggregate net of   20 xs 0
agg J.Re13  5 claims 100 xs 0 dsev [1:100]          occurrence net of   15 xs 5␣
↪poisson aggregate ceded to 20 xs 0
agg J.Re14  1000 prem at .5 lr sev lognorm 10 cv [.2 .35 .5] wts=3 occurrence␣
↪ceded to .5 xs .5 and .5 po 1.0 xs 1.0 and 25% so 3.0 xs 2.0       mixed gamma 0.
↪5  aggregate ceded to 1.5 xs 1.5 and 5.0 po 10.0 xs 12.5
agg J.Re15  10 claims          sev lognorm  3 cv 0.35             occurrence net␣
↪of 1 xs 1 and 2.5 po 5 xs 5 and 25% so 30 xs 20 and inf xs 100 poisson        ␣
↪aggregate net of 10 xs 15
agg J.Re16 5 claims dsev [0:100] occurrence ceded to tower [10:50:10] poisson
agg J.Re17 5 claims dsev [0:100] poisson aggregate ceded to tower [200:500:100]

# Tweedie distributions
# =====================
agg K.Tweedie0 10.050251256281404 claims sev gamma 0.0995 cv 0.0708812050083283␣
↪poisson note{Tweedie defined by claim count, gamma mean and cv}
agg K.Tweedie1 10.050251256281404 claims sev 0.0005 * gamma 199 poisson         ␣
↪      note{Tweedie defined by claim count, gamma scale and shape}
agg K.Tweedie2 tweedie 1 1.005 0.1                                              ␣
↪      note{Tweedie defined using mean, p, and dispersion, variance = dispersion␣
↪xs mean**p}

# Examples from books and papers
# ==============================
# The way this file is interpreted by the deubugger forces the programs to be␣
↪written on one line
port L.Bodoff1 note{Bodoff Thought Experiment No. 1} agg wind1  1 claim sev␣
↪dhistogram xps [0,  99] [0.80, 0.20] fixed agg quake1 1 claim sev dhistogram xps␣
↪[0, 100] [0.95, 0.05] fixed

port L.Bodoff2 note{Bodoff Thought Experiment No. 2} agg wind2  1 claim sev␣
↪dhistogram xps [0,  50] [0.80, 0.20] fixed agg quake2 1 claim sev dhistogram xps␣
↪[0, 100] [0.95, 0.05] fixed

port L.Bodoff3 note{Bodoff Thought Experiment No. 3} agg wind3  1 claim sev␣
↪dhistogram xps [0,   5] [0.80, 0.20] fixed agg quake3 1 claim sev dhistogram xps␣
↪[0, 100] [0.95, 0.05] fixed

port L.Bodoff4 note{Bodoff Thought Experiment No. 4 (check!)} agg a 0.25 claims␣
↪sev   4 * expon poisson agg b 0.05 claims sev  20 * expon poisson agg c 0.05␣
↪claims sev 100 * expon poisson

# PIR book case studies
# =====================
# hints: reg_p=1; roe=0.10;
# numbers in names ensure correct sort order
port M.PIR.1.Discrete note{PIR Discrete case study. Change 8 to 9 for the equal␣
↪points example.} agg Discrete.X1 1 claim dsev [0 8 10] [1/2 1/4 1/4] fixed agg␣
↪Discrete.X2 1 claim dsev [0 1 90] [1/2 1/4 1/4] fixed note{bs=1; log2=8;␣
↪padding=1}

# hints: reg_p=.999, roe=0.10
port M.PIR.2.Tame note{PIR Tame case study. For reinsurance see text.} agg Tame.A␣
↪1 claim sev gamma  50 cv 0.10 fixed agg Tame.B 1 claim sev gamma  50 cv 0.15␣
↪fixed note{bs=1/64, log2=16, padding=1}

# hints: reg_p=.999, roe=0.10
port M.PIR.3.CNC.Gross note{PIR Cat-Noncat gross case study.} agg CNC.NonCat 1␣
↪claim sev gamma    80 cv 0.15 fixed agg CNC.Cat 1 claim sev lognorm  20 cv 1.00␣
```

(continues on next page)

(continued from previous page)

```
↪fixed note{bs=1/64, log2=16, padding=1}

port M.PIR.4.CNC.Net note{PIR Cat-Noncat net case study.} agg CNC.Net.NonCat 1␣
↪claim sev gamma    80 cv 0.15 fixed agg CNC.Net.Cat    1 claim sev lognorm  20␣
↪cv 1.00 fixed aggregate net of 79.64 xs 41.11 note{bs=1/64, log2=16, padding=1}

# hints: reg_p=.999, roe=0.10,
port M.PIR.5.HuSCS.Gross note{PIR Hurricane SCS gross case study.} agg SCS 70␣
↪claims sev exp(-1.9**2 / 2)      * lognorm 1.9 poisson agg Hu   2 claims sev␣
↪exp(-2.5**2/2)/(1/15) * lognorm 2.5 poisson note{bs=1/4, log2=19, padding=1}

port M.PIR.6.HuSCS.Net note{PIR Hurricane SCS net case study.} agg Net.SCS 70␣
↪claims sev    exp(-1.9**2 / 2)    * lognorm 1.9 poisson agg Net.Hu   2 claims sev␣
↪exp(-2.5**2/2)/(1/15) * lognorm 2.5 occurrence net of 372.4 xs 40.25 poisson note
↪{bs=1/4, log2=19, padding=1}

# Proxies for US Lines
# ====================
# for portfolio construction: 10M sized books with "reasonable" severity curves
# IRS: used cycle adjusted numbers, no further adjustments
# RMI: see  \S\Teaching\2019-09\RMI3388\Notes\pdf\Archive\QA16_RMI3388_Tue-29-Oct-
↪2019_cat_models.pdf
# SCS is a swag; last pages of 2021 cat report for EF3+ tornados shows 25-50/year;␣
↪big
# SCS events 3-6 billion; https://www.aon.com/reinsurance/catastropheinsight/
↪global-regional-losses.html?region=United%20States&type=insured
# shows avg / year about 25B, so mean severity is 25 / 37.5 = 2/3 = 666M
# sigma 1.75 of  makes 10B a 1 in 16 event; mu mean=666 eq np.log(666e6) - 1.75**2/
↪2 = 18.785550228504665
agg N.US.Hurricane 1.79 claims 1e12 xs 0 sev exp(19.595) * lognorm 2.581 poisson ␣
↪note{Based on NOAA reanalysis dataset and sample of hurricane loss. Source SJU␣
↪RMI3388 Course.}
agg N.US.SCS      37.50 claims 50e9 xs 0 sev exp(18.785550228504665) * lognorm 1.
↪75 poisson note{Judgmental based on fre3quency of EF4-5 tornadoes, AAL of 25B␣
↪and extreme event loss of 10B}

# CVs based on sigmas of 1.5 for liab, 1.75 for prof and 1.25 for ppa
# Average severity selected judgmentally
agg N.Comm.Liability     10e6 loss 1e6 xs 0   sev lognorm 100e3 cv 3.77 mixed␣
↪gamma 0.26 note{Source: CV from Aon IRS 10th ed, severity selected judgmentally.}
agg N.Comm.Professional  10e6 loss 5e6 xs 0   sev lognorm 850e3 cv 8.48 mixed␣
↪gamma 0.27 note{Source: CV from Aon IRS 10th ed, severity selected judgmentally.}
agg N.Personal.Auto          10e6 loss 300e3 xs 0 sev lognorm  45e3 cv 20.4␣
↪mixed gamma 0.13 note{Source: CV from Aon IRS 10th ed, severity selected␣
↪judgmentally.}

# limits profile low and high limits and attaching book: 1/1, 4/1 and 5/5
# severity midway between liability and professional
agg N.Comm.Umbrella [6e6 3e6 1e6] loss [1e6 4e6 5e6] xs [1e6 1e6 5e6] sev lognorm␣
↪250e3 cv 5.00 mixed gamma 0.26 note{Source: CV from Aon IRS 10th ed, severity␣
↪selected judgmentally.}

# property mixed severity property and cat, pass through mixing
# note on CV of betas: for high cv it becomes a zero / one variable, to the␣
↪variance maxes out at p (1-p) where p is the mean
# thus the CV maxes out at 1/sqrt(p(1-p)). For a mean of 0.1 the variance equals 1/
↪sqrt(0.09) = 0.3 and the cv is 0.3 / 0.1 = 3
# could limit profile too, but keep simple
# assumes full ITV
agg N.Comm.Property [7e6 3e6] loss 100e6 xs 0 sev 100e6 * beta 0.1 cv [1.25 2.5]␣
↪mixed gamma 0.28 note{Source: CV from Aon IRS 10th ed, severity selected␣
```

(continues on next page)

(continued from previous page)

```
↪judgmentally.}
agg N.Homeowners           [8e6 2e6] loss 650e3 xs 0 sev 650e3 * beta 0.1 cv [0.75␣
↪2.75] mixed gamma 0.37 note{Source: CV from Aon IRS 10th ed, severity selected␣
↪judgmentally.}

# really a severity curve, but you can't mix severity
# agg N.Comm.Auto.MED 1 claim sev [2.764e3 24.548e3 275.654e3 1.917469e6 10e6] *␣
↪expon 1 wts [0.824796 0.159065 0.014444 0.001624 0.000071] fixed note{Use␣
↪log2=18 and bs=500 or apply limits. Source: Example from Mixed Exponential␣
↪snippet and Similar Risks IME paper (2022).; log2=18; bs=500;}
sev N.Comm.Auto.MED [2.764e3 24.548e3 275.654e3 1.917469e6 10e6] * expon 1 wts [0.
↪824796 0.159065 0.014444 0.001624 0.000071] note{Source: Example from Mixed␣
↪Exponential snippet and Similar Risks IME paper (2022)}

# Novelties
# =========
agg O.Logo 1 claim [10:250:10] xs 0 sev lognorm 100 cv 1 fixed
```

To only parse:

```
from aggregate import build
filename = build.default_dir / 'test_suite.agg'
assert filename.exists()

build.logger_level(30)
df = build.interpreter_file(filename=filename)

df.query('error != 0')
```

4.5 `sly` Parser

The parser is built using the `sly` package, https://sly.readthedocs.io/en/latest/sly.html.

CHAPTER

FIVE

TECHNICAL GUIDES

Technical Guides cover theory and implementation. How calculations work in theory and how they have been implemented in `aggregate`.

5.1 Probability Background

Objectives: Statement and limited explanation of important probability concepts that underlie `aggregate` calculations.

Audience: Readers looking for a probability refresher.

Prerequisites: Knowledge of basic probability and calculus (real analysis).

See also: *Insurance Probability*.

Notation: The variance of a random variable X is $\mathsf{var}(X)=\mathsf{E}[X^2]-\mathsf{E}[X]^2$. The standard deviation is $\sigma(X)=\sqrt{\mathsf{var}(X)}$. The coefficient of variation (CV) of X is $\mathsf{CV}(X)=\sigma(X)/\mathsf{E}[X]$. The skewness of X is $\mathsf{E}[(X-\mathsf{E}[X])^3]/\sigma(X)^3$.

Contents:

5.1.1 Helpful References

- Klugman *et al.* [2019]
- Panjer and Willmot [1992]
- Williams [1991]
- Feller [1971]
- Loeve [2017]
- Johnson *et al.* [2005]

- Mildenhall [2017]

5.1.2 Types

Todo: Documentation to follow.

5.1.3 Severity Distributions

Computing moments

Higher moments of a layer with a limit y excess of an attachment (deductible, retention) a can be computed as

$$\begin{aligned}\mathsf{E}[((X-a)^+\wedge l)^n] &= \int_a^{a+l}(x-a)^n f(x)\,dx + l^n S(a+l) \\ &= \sum_{k=0}^{n}(-1)^k\binom{n}{k}a^{n-k}\int_a^{a+l}x^k f(x)\,dx + l^n S(a+l) \\ &= \sum_{k=0}^{n}(-1)^k\binom{n}{k}a^{n-k}\left(\mathsf{E}[k;a+l]-\mathsf{E}[k;a]\right) + l^n S(a+l)\end{aligned}$$

where

$$\mathsf{E}[k;a] = \int_0^a x^k f(x)\,dx$$

is the partial expectation function.

Lognormal

For the lognormal, the trick for higher moments is to observe that if X is lognormal (μ,σ) then X^k is lognormal $(k\mu,k\sigma)$. The formula for partial expectations of the lognormal is easy to compute by substitution, giving

$$\mathsf{E}[k,a] = \exp(k\mu + k^2\sigma^2/2)\Phi\left(\frac{\log x-\mu-k\sigma^2}{\sigma}\right)$$

Densities of the form $f(x)=x^\alpha c(\alpha)g(x)$

$$\begin{aligned}\mathsf{E}[k,a] &= \int_0^a x^k x^\alpha c(\alpha)g(x)\,dx \\ &= \frac{c(\alpha)}{c(n+\alpha)}\int_0^a x^{k+\alpha}c(k+\alpha)g(x)\,dx \\ &= \frac{c(\alpha)}{c(n+\alpha)}F_{k+\alpha}(a)\end{aligned}$$

are easy to express in terms of the distribution function. This is a broad class including the gamma.

Pareto

An easy integral computation, substitute $y = \lambda + x$ to express in powers of y:

$$\begin{aligned}
\mathsf{E}[k,a] &= \int_0^a \alpha x^k \frac{\lambda^\alpha}{(\lambda+x)^{\alpha+1}}\,dx \\
&= \int_\lambda^{\lambda+a} \alpha\lambda^\alpha \frac{(y-\lambda)^k}{y^{\alpha+1}}\,dy \\
&= \sum_{i=0}^k (-1)^{k-i}\alpha\lambda^\alpha \binom{k}{i} \int_\lambda^{\lambda+a} y^{i-\alpha-1}\lambda^{k-i}\,dy \\
&= \sum_{i=0}^k (-1)^{k-i}\alpha\lambda^{\alpha+k-i} \binom{k}{i} \frac{y^{i-\alpha}}{i-\alpha}\Big|_\lambda^{\lambda+a}.
\end{aligned}$$

`scipy.stats` Severity Distributions

All zero, one, and two shape parameter `scipy.stats` continuous random variable classes can be used as severity distributions. See list of distributions for details about each available option.

5.1.4 Frequency Distributions

The following reference is from the `scipy` documentation.

Bernoulli Distribution

The probability mass function for *bernoulli* is:

$$f(k) = \begin{cases} 1-p & \text{if } k = 0 \\ p & \text{if } k = 1 \end{cases}$$

for k in $\{0, 1\}$, $0 \leq p \leq 1$ *bernoulli* takes p as shape parameter, where p is the probability of a single success and $1-p$ is the probability of a single failure.

Binomial Distribution

The probability mass function for *binom* is:

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

for $k \in \{0, 1, \dots, n\}$, $0 \leq p \leq 1$ *binom* takes n and p as shape parameters, where p is the probability of a single success and $1-p$ is the probability of a single failure.

Geometric Distribution

The probability mass function for *geom* is:

$$f(k) = (1-p)^{k-1}p$$

for $k \ge 1$, $0 < p \le 1$ *geom* takes p as shape parameter, where p is the probability of a single success and $1-p$ is the probability of a single failure.

Poisson Distribution

The probability mass function for *poisson* is:

$$f(k) = \exp(-\mu)\frac{\mu^k}{k!}$$

for $k \ge 0$. *poisson* takes $\mu \ge 0$ as shape parameter.

Neyman (A) Distribution

The Neyman distribution is a Poisson stopped-sum distribution of Poisson variables, see Johnson *et al.* [2005].

Fixed Distribution

The fixed distribution takes a single value with probability one.

5.1.5 Moment Generating Functions

The moment generating function of a random variable X is defined as

$$M_X(z) = \mathsf{E}[\exp(zX)].$$

The moment generating function is related to the characteristic function of X which is defined as $\phi_X(z) = \mathsf{E}[\exp(izX)] = M_X(iz)$. ϕ is guaranteed to converge for all real z and so is preferred in certain situations.

Moment generating functions get their name from the fundamental property that

$$\frac{\partial^n M_X}{\partial z^n}\Big|_{z=0} = \mathsf{E}[X^n]$$

for all positive integers n provided the differential exists.

Let F be the distribution function of X. Feller [1971] Section XVII.2a shows that if F has expectation μ then ϕ, the characteristic function of F, has a derivative ϕ' and $\phi'(0) = i\mu$. However the converse is false. Pitman proved that the following are equivalent.

1. $\phi'(0) = i\mu$.
2. As $t \to \infty$, $t(1 - F(t) + F(-t)) \to 0$ and

   $$\int_t^{-t} x dF(x) \to \mu,$$

   where $F(-t) := \lim F(s)$ as $s \uparrow t$.

3. The average $(X_1 + \cdots + X_n)/n$ tends in probability to μ, that is $\mathsf{Pr}(|(X_1 + \cdots + X_n)/n - \mu| > \epsilon) \to 0$ as $n \to \infty$.

The condition for the limit in 2 to exist is weaker than the requirement that $\mathsf{E}[X]$ exists if X is supported on the whole real line. For the expectation to exist requires $\int_{-\infty}^{\infty} xdF(x)$ exists which means $\lim_{t\to-\infty}\lim_{s\to\infty}\int_t^s xdF(x)$.

The moment generating function of a bivariate distribution (X_1, X_2) is defined as

$$M_{X_1,X_2}(z_1, z_2) = \mathsf{E}[\exp(z_1X_1 + z_2X_2)].$$

It has the property that

$$\left.\frac{\partial^{m+n} M_{X_1,X_2}}{\partial z_1^m \partial z_2^n}\right|_{(0,0)} = \mathsf{E}[X_1^m X_2^n]$$

for all positive integers n, m.

The MGF of a normal variable with mean μ and standard deviation σ is

$$M(z) = \exp(\mu z \sigma^2 z^2/2).$$

The MGF of a Poisson variable with mean n is

$$M(z) = \exp(n(e^z - 1)).$$

See any standard text on probability for more information on moment generating functions, characteristic functions and modes of convergence.

5.1.6 Mixed Frequency Distributions

A random variable N is G-mixed Poisson if $N \mid G$ has a Poisson nG distribution for some fixed non-negative n and a non-negative mixing distribution G with $\mathsf{E}[G] = 1$. Let $\mathsf{var}(G) = c$ and let $\mathsf{E}[G^3] = g$. Glenn Meyers calls c the contagion.

The MGF of a G-mixed Poisson is

$$M_N(z) = \mathsf{E}[e^{zN}] = \mathsf{E}[\mathsf{E}[e^{zN} \mid G]] = \mathsf{E}[e^{nG(e^z-1]}) = M_G(n(e^z - 1))$$

since $M_G(z) := \mathsf{E}[e^{zG}]$ and the MGF of a Poisson with mean n is $\exp(n(e^z - 1))$. Thus

$$\mathsf{E}[N] = M_N'(0) = nM_G'(0) = n,$$

because $\mathsf{E}[G] = M_G'(0) = 1$. Similarly

$$\mathsf{E}[N^2] = M_N''(0) = n^2 M_G''(0) + nM_G'(0) = n^2(1 + c) + n$$

and so

$$\mathsf{var}(N) = n(1 + cn).$$

Finally

$$\begin{aligned}\mathsf{E}[N^3] &= M_N'''(0)\\ &= n^3 M_G'''(0) + 3n^2 M_G''(0) + nM_G'(0) = gn^3 + 3n^2(1 + c) + n\end{aligned}$$

and therefore the central moment

$$\mathsf{E}(N - \mathsf{E}[N])^3 = n^3(g - 3c - 1) + 3cn^2 + n.$$

We can also assume G has mean n and work directly with G rather than nG, $\mathsf{E}[G] = 1$. We will call both forms mixing distributions.

Gamma Mixing

A negative binomial is a gamma-mixed Poisson: if $N \mid G$ is distributed as a Poisson with mean G, and G has a gamma distribution, then the unconditional distribution of N is a negative binomial. A gamma distribution has a shape parameter a and a scale parameter θ so that the density is proportional to $x^{a-1}e^{x/\theta}$, $\mathsf{E}[G] = a\theta$ and $\mathsf{var}(G) = a\theta^2$.

Let $c = \mathsf{var}(G) = \nu^2$, so ν is the coefficient of variation of the mixing distribution. Then

- $a\theta = 1$ and $a\theta^2 = c$
- $\theta = c = \nu^2$, $a = 1/c$

The non-central moments of the gamma distribution are $\mathsf{E}[G^r] = \theta^r\Gamma(a+r)/\Gamma(a)$. Therefore $Var(G) = a\theta^2$ and $E(G - E(G))^3 = 2a\theta^3$. The skewness of G is $\gamma = 2/\sqrt(a) = 2\nu$.

Applying the general formula for the third central moment of N we get an expression for the skewness

$$\mathsf{skew}(N) = \frac{n^3(\gamma - 3c - 1) + n^2(3c+2) + n}{(n(1+cn))^{3/2}}.$$

The corresponding MGF of the gamma is $M_G(z) = (1 - \theta z)^{-a}$.

The gamma and negative binomial occur in the literature with many different parameterizations. The main ones are shown in the next three tables.

Table 1: Parameterizations of the Gamma Distribution

Model	Density	MGF	Mean	Var
(a) α, β	$\dfrac{x^{\alpha-1}e^{-x/\beta}}{\beta^\alpha\Gamma(\alpha)}$	$(1-\beta t)^{-\alpha}$	$\alpha\beta$	$\alpha\beta^2$
(b) α, β	$\dfrac{x^{\alpha-1}\beta^\alpha e^{-x\beta}}{\Gamma(\alpha)}$	$(1-t/\beta)^{-\alpha}$	α/β	α/β^2
(c) α, θ	$\dfrac{x^{\alpha-1}e^{-x/\theta}}{\theta^\alpha\Gamma(\alpha)}$	$(1-t\theta)^{-\alpha}$	$\alpha\theta$	$\alpha\theta^2$

Model (a) is used by Microsoft Excel, Wang [1998], and Johnson *et al.* [2005] Chapter 17. Model (b) is used by Bowers *et al.* [1997]. Model (c) is used by . Obviously model (c) is just model (a) with a change of notation.

Table 2: Parameterizations of the Negative Binomial Distribution

Model	Density	MGF	Mean	Var
(a) α, β	$\binom{\alpha+x-1}{x}\left(\frac{\beta}{1+\beta}\right)^x\left(\frac{1}{1+\beta}\right)^\alpha$	$(1-\beta(e^t-1))^{-\alpha}$	$\alpha\beta$	$\alpha\beta^2$
(b) P, k	$\binom{k+x-1}{x}\left(\frac{P}{Q}\right)^x\left(\frac{Q-P}{Q}\right)^k$	$(Q-Pe^t)^{-k}$	kP	kPQ
(c) $p, r > 0$	$\binom{r+x-1}{x}p^r q^x$	$\dfrac{p^r}{(1-qe^s)^r}$	rq/p	rq/p^2

Note that $Q = P + 1$, $q = 1 - p$, $0 < p < 1$ and $r > 0$, and $P = 1/(\beta + 1)$.

Table 3: Fitting the Negative Binomial Distribution

Model	Parameters	VM Scale	VM Shape	Ctg Scale	Ctg Shape
(a)	r, β	$r = m/(v-1)$	$\beta = v - 1$	$r = 1/c$	$\beta = cn$
(b)	k, P	$k = m/(v-1)$	$P = v - 1$	$k = 1/c$	$P = cn$
(c)	r, p	$r = m/(v-1)$	$p = 1/v$	$r = 1/c$	$p = 1/(1+cn)$

In model (c) the parameter r need not be an integer because the binomial coefficient can be computed as

$$\binom{r+x-1}{x} = \frac{\Gamma(r+x)}{\Gamma(r)x!},$$

an expression which is valid for all r. The cumulative distribution function of the negative binomial can be computed using the cumulative distribution of the beta distribution. Using the model (c) parameterization, if N is negative binomial p, r then

$$\begin{aligned}\mathsf{Pr}(N \le k) &= \text{BETADIST}(p; r, k+1) \\ &:= \frac{1}{B(r, k+1)} \int_0^p u^{r-1}(1-u)^k du\end{aligned}$$

where B is the complete beta function. See Johnson, Kotz and Kemp [Eqn. 5.31] for a derivation. BETADIST is the Excel beta cumulative distribution function.

The name negative binomial comes from an analogy with the binomial. A binomial variable has parameters n and p, mean np and variance npq, where $p+q=1$. It is a sum of n independent Bernoulli variables B where $\mathsf{Pr}(B=1)=p$ and $\mathsf{Pr}(B=0)=q=1-p$. The MGF for a binomial is $(q+pe^z)^n$ and the probabilities are derived from the binomial expansion of the MGF. By analogy the negative binomial can be defined in terms of the negative binomial expansion of $(Q-Pe^z)^{-k}$ where $Q=1+P$, $P>0$ and $k>0$.

The actuary can look at the negative binomial in two different way, each of which gives different results. It is important to understand these two views. First there is the **contagion view**, where the mixing distribution G has mean n and variance c producing a negative binomial with mean n and variance $n(1+cn)$. (In fact G is a gamma with model (a) parameters $\alpha = r$ and $\beta = 1/r$.) The word contagion is used by Heckman and Meyers [1983] and is supposed to indicate a “contagion” of claim propensity driven by common shock uncertainty, such as claim inflation, economic activity, or weather. Here the variance grows with the square of n and the coefficient of variation tends to $\sqrt{c} > 0$ as $n \to \infty$. Secondly, one can consider an over-dispersed family of Poisson variables with mean n and variance vn for some $v > 1$. We call v the **variance multiplier**. Now, the coefficient of variation tends to 0 as $n \to \infty$. The notion of over-dispersion and its application in modeling is discussed in Clark and Thayer [2004] and Verrall [2004].

The Variance Multiplier

The variance of a mixed Poisson equals $n(1+cn)$ where c equals the variance of the mixing distribution. Thus the variance equals $v = 1 + cn$ times the mean n, where v is called the **variance multiplier**. The variance multiplier specification is used by some US rating bureaus. The dictionary to variance and mix CV is

$$\begin{aligned}c &= (v-1)/n \\ cv &= \sqrt{(v-1)/n}.\end{aligned}$$

The frequency for an excess layer attaching at a equals $nS(a)$. For fixed c, the implied variance multiplier $v = 1 + cnS(a)$ decreases and the excess claim count distribution converges to a Poisson. This is an example of the law of small numbers.

Per Mildenhall [2017], if ν is the CV of G then the ν equals the asymptotic coefficient of variation for any G-mixed compound Poisson distribution whose variance exists. The variance will exist iff the variance of the severity term exists. See 5_x_severity_irrelevant.

Negative Binomial Distribution

Negative binomial distribution describes a sequence of iid Bernoulli trials, repeated until a predefined, non-random number of successes occurs.

The probability mass function of the number of failures for *nbinom* is:

$$f(k) = \binom{k+n-1}{n-1} p^n (1-p)^k$$

for $k \ge 0$, $0 < p \le 1$

nbinom takes n and p as shape parameters where n is the number of successes, p is the probability of a single success, and $1-p$ is the probability of a single failure.

Another common parameterization of the negative binomial distribution is in terms of the mean number of failures μ to achieve n successes. The mean μ is related to the probability of success as

$$p = \frac{n}{n+\mu}$$

The number of successes n may also be specified in terms of a "dispersion", "heterogeneity", or "aggregation" parameter α, which relates the mean μ to the variance σ^2, e.g. $\sigma^2 = \mu + \alpha\mu^2$. Regardless of the convention used for α,

$$p = \frac{\mu}{\sigma^2}$$
$$n = \frac{\mu^2}{\sigma^2 - \mu}$$

Beta Binomial Distribution

The beta-binomial distribution is a binomial distribution with a probability of success p that follows a beta distribution.

The probability mass function for *betabinom* is:

$$f(k) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}$$

for $k \in \{0, 1, \dots, n\}$, $n \ge 0$, $a > 0$, $b > 0$, where $B(a,b)$ is the beta function.

betabinom takes n, a, and b as shape parameters.

Shifted Mixing (General)

We can adjust the skewness of mixing with shifting. In addition to a target CV ν assume a proportion f of claims are sure to occur. Use a mixing distribution $G = f + G'$ such that

- $\mathsf{E}[G] = f + \mathsf{E}[G'] = 1$ and
- $\mathsf{CV}(G) = \sigma(G') = \nu$.

As f increases from 0 to 1 the skewness of G will increase. Delaporte first introduced this idea.

Since $\mathsf{skew}(G) = \mathsf{skew}(G')$ we have $g = \mathsf{E}[G^3] = \nu^3\mathsf{skew}(G') + 3c + 1$.

Delaporte Mixing (Shifted Gamma)

Inputs are target CV ν and proportion of certain claims f, $0\le f\le 1$. Find parameters f, a and θ for a shifted gamma $G=f+G'$ with $E(G')=1-f$ and $SD(G')=\nu$ as

- f is input
- mean $a\theta=1-s$ and $CV=\nu=\sqrt{a}\theta$ so $a=(1-f)^2/\nu^2=(1-f)^2/c$ and $\theta=(1-f)/a$

The skewness of G equals the skewness of G' equals $2/\sqrt{a}=2\nu/(1-f)$, which is then greater than the skewness 2ν when $f=0$. The third non-central moment $g=2\nu^4/(1-f)+3c+1$

Poisson Inverse Gaussian Distribution

The $(a,b,0)$ and $(a,b,1)$ Classes

See Klugman *et al.* [2019].

5.1.7 Aggregate Distributions

Let $A=X_1+\cdots+X_N$ be an aggregate distribution, where N is the **frequency** component and X_i are iid **severity** random variables.

Aggregate Mean

The mean of a sum equals the sum of the means. Let $A=X_1+\cdots+X_N$. If $N=n$ is fixed then $\mathsf{E}[A]=n\mathsf{E}[X]$, because all $\mathsf{E}[X_i]=\mathsf{E}[X]$. In general,

$$\mathsf{E}[A]=\mathsf{E}[X]\mathsf{E}[N]$$

by conditional probability.

Aggregate Variance

The variance of a sum of independent random variables equals the sum of the variances. If $N=n$ is fixed then $\mathsf{Var}(A)=n\mathsf{Var}(X)$ and $\mathsf{Var}(N)=0$. If $X=x$ is fixed then $\mathsf{Var}(A)=x^2\mathsf{Var}(N)$ and $\mathsf{Var}(X)=0$. Making the obvious associations $n\leftrightarrow\mathsf{E}[N]$, $x\leftrightarrow\mathsf{E}[X]$ suggests

$$\mathsf{Var}(A)=\mathsf{E}[N]\mathsf{Var}(X)+\mathsf{E}[X]^2\mathsf{Var}(N).$$

Using conditional expectations and conditioning on the value of N shows this is the correct answer!

Exercise. Confirm the formulas for an aggregate mean and variance hold for the *Simple Example*.

Aggregate Moment Generating Function

Using the tower property of conditional expectations and the independence of N and X_i gives

$$\begin{aligned}
M_A(z) &= \mathsf{E}[\exp(z(X_1+\cdots X_N))]\\
&= \mathsf{E}[\mathsf{E}[\exp(z(X_1+\cdots X_N))\mid N]]\\
&= \mathsf{E}[\mathsf{E}[\exp(zX_1)^N]]\\
&= \mathsf{E}[\mathsf{E}[\exp(zX_1)]^N]\\
&= M_N(\log(M_X(z)))
\end{aligned}$$

Differentiating and using XXs formula, yields the moments of A, see below.

The last expression is very important and underlies the use of FFTs to compute aggregate distributions.

Next, specialize to the case where $A = X_1 + \cdots + X_N$ is an aggregate distribution with N a G-mixed Poisson. Then

$$\begin{aligned} M_A(z) &= \mathsf{E}[\exp(z(X_1 + \cdots X_N))] \\ &= \mathsf{E}[\mathsf{E}[\exp(z(X_1 + \cdots X_N)) \mid N]] \\ &= \mathsf{E}[\mathsf{E}[\exp(zX_1)^N]] \\ &= \mathsf{E}[\mathsf{E}[M_X(z)^N \mid G]] \\ &= \mathsf{E}[\exp(nG(M_X(z) - 1))] \\ &= M_G(n(M_X(z) - 1)) \end{aligned}$$

Thus

$$\mathsf{E}[A] = M_A'(0) = nM_G'(0)M_X'(0) = n\mathsf{E}[X]$$

and

$$\begin{aligned} \mathsf{E}[A^2] &= M_A''(0) \\ &= n^2 M_G''(0)M_X'(0)^2 + nM_G'(0)M_X''(0) \\ &= n^2\mathsf{E}[G^2]\mathsf{E}[X]^2 + n\mathsf{E}[X^2]. \end{aligned}$$

Hence, using the fact that $\mathsf{E}[G^2] = 1 + c$,

we get

$$\begin{aligned} \mathsf{var}(A) &= n^2\mathsf{E}[G^2]\mathsf{E}[X]^2 + n\mathsf{E}[X^2] - n^2\mathsf{E}[X]^2 \\ &= n^2c\mathsf{E}[X]^2 + n\mathsf{E}[X^2] \\ &= (\mathsf{var}(N) - \mathsf{E}[N])\mathsf{E}[X]^2 + \mathsf{E}[N]\mathsf{E}[X^2] \\ &= \mathsf{var}(N)\mathsf{E}[X]^2 + \mathsf{E}[N]\mathsf{var}(X). \end{aligned}$$

Continuing along the same vein we get

$$\begin{aligned} \mathsf{E}[A^3] =& \mathsf{E}[N]\mathsf{E}[X^3] + \mathsf{E}[N^3]\mathsf{E}[X]^3 + 3\mathsf{E}[N^2]\mathsf{E}[X]\mathsf{E}[X^2] \\ &- 3\mathsf{E}[N]\mathsf{E}[X]\mathsf{E}[X^2] - 3\mathsf{E}[N^2]\mathsf{E}[X]^3 + 2\mathsf{E}[N]\mathsf{E}[X]^3. \end{aligned}$$

and so we can compute the skewness of A, remembering that

$$\mathsf{E}[(A - \mathsf{E}[A])^3] = \mathsf{E}[A^3] - 3\mathsf{E}[A^2]\mathsf{E}[A] + 2\mathsf{E}[A]^3.$$

Further moments can be computed using derivatives of the moment generating function.

Having computed the mean, CV and skewness of the aggregate using these equations we can use the method of moments to fit a shifted lognormal or shifted gamma distribution. We turn next to a description of these handy distributions.

5.1.8 Shifted Gamma and Lognormal Distributions

The shifted gamma and shifted lognormal distributions are versatile three parameter distributions whose method of moments parameters can be conveniently computed by closed formula. The examples below show that they also provide a very good approximation to aggregate loss distributions. The shifted gamma approximation to an aggregate is discussed in Bowers *et al.* [1997]. Properties of the shifted gamma and lognormal distributions, including the method of moments fit parameters, are also shown in Daykin *et al.* [1993] chapter 3.

Let L have a lognormal distribution. Then $S = s \pm L$ is a shifted lognormal, where s is a real number. Since s can be positive or negative and since L can equal $s + L$ or $s - L$, the shifted lognormal can model distributions which are positively or negatively skewed, as well as distributions supported on the negative reals. The key facts about the shifted lognormal are shown in Table 1.4. The variable η is a solution to the cubic equation

$$\eta^3 + 3\eta - \gamma = 0$$

where γ is the skewness.

Let G have a gamma distribution. Then $T = s \pm G$ is a shifted gamma distribution, where s is a real number. Table 1.1 shows some common parametric forms for the gamma distribution. The key facts about the shifted gamma distribution are also shown in Table 1.4.

The exponential is a special case of the gamma where $\alpha = 1$. The χ^2 is a special case where $\alpha = k/2$ and $\beta = 2$ in the Excel parameterization. The Pareto is a mixture of exponentials where the mixing distribution is gamma.

Table 4: Shifted Gamma and Lognormal Distributions

Item	Shifted Gamma	Shifted Lognormal
Parameters	s, α, θ	s, μ, σ
Mean m	$s + \alpha\theta$	$s + \exp(\mu + \sigma^2/2)$
Variance	$\alpha\theta^2$	$m^2 \exp(\sigma^2 - 1)$
CV, ν	$\sqrt{\alpha}\beta/\gamma$	$\exp((\sigma^2 - 1)/2)$
Skewness,	$2/\sqrt{\alpha}$	$\gamma = \nu(\nu^2 + 3)$
Method of Moments Parameters		
η	n/a	$\eta = u - 1/u$ where
		$u^3 = \sqrt{\gamma^2 + 4}/2 + \gamma/2$
Shift variable, s	$m - \alpha\beta$	$m(1 - \nu\eta)$
α or σ	$4/\gamma^2$	$\sqrt{\ln(1 + \eta^2)}$
β or μ	$m\nu\gamma/2$	$\ln(m - s) - \sigma^2/2$

5.1.9 Appendix: Selected `scipy.stats` Discrete Random Variables

Here is the list of `scipy.stats` discrete random variables.

```
                         Num. args   Min range   Max range Parameters
Distribution
bernoulli                        1           0           1 'p'
dlaplace                         1        -inf         inf 'a'
geom                             1           1         inf 'p'
logser                           1           1         inf 'p'
planck                           1           0         inf 'lambda\_'
poisson                          1           0         inf 'mu'
yulesimon                        1           1         inf 'alpha'
zipf                             1           1         inf 'a'
binom                            2           0         inf 'n' and 'p'
boltzmann                        2           0         inf 'lambda\_' and 'N'
nbinom                           2           0         inf 'n' and 'p'
randint                          2           0         inf 'low' and 'high'
skellam                          2        -inf         inf 'mu1' and 'mu2'
zipfian                          2           1         inf 'a' and 'n'
betabinom                        3           0         inf 'n', 'a', and 'b'
hypergeom                        3           0         inf 'M', 'n', and 'N'
nhypergeom                       3           0         inf 'M', 'n', and 'r'
nchypergeom_fisher               4           0         inf 'M', 'n', 'N', and
↪'odds'
nchypergeom_wallenius            4           0         inf 'M', 'n', 'N', and
↪'odds'
```

- `bernoulli` **Bernoulli** (help). The probability mass function for *bernoulli* is:

$$f(k) = \begin{cases} 1 - p & \text{if } k = 0 \\ p & \text{if } k = 1 \end{cases}$$

  for k in $\{0, 1\}$, $0 \le p \le 1$

bernoulli takes p as shape parameter, where p is the probability of a single success and $1-p$ is the probability of a single failure.

The probability mass function above is defined in the "standardized" form. To shift distribution use the `loc` parameter. Specifically, `bernoulli.pmf(k, p, loc)` is identically equivalent to `bernoulli.pmf(k - loc, p)`.

- **`betabinom` Betabinom (help). The beta-binomial distribution is a binomial distribution with a** probability of success *p* that follows a beta distribution.

  The probability mass function for *betabinom* is:

  $$f(k) = \binom{n}{k} \frac{B(k+a, n-k+b)}{B(a,b)}$$

  for $k \in \{0, 1, \dots, n\}$, $n \geq 0$, $a > 0$, $b > 0$, where $B(a, b)$ is the beta function.

  betabinom takes n, a, and b as shape parameters.

- `binom` **Binom** (help). The probability mass function for *binom* is:

  $$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

  for $k \in \{0, 1, \dots, n\}$, $0 \leq p \leq 1$

  binom takes n and p as shape parameters, where p is the probability of a single success and $1-p$ is the probability of a single failure.

- `boltzmann` **Boltzmann** (help). The probability mass function for *boltzmann* is:

  $$f(k) = (1 - \exp(-\lambda)) \exp(-\lambda k) / (1 - \exp(-\lambda N))$$

  for $k = 0, ..., N-1$.

  boltzmann takes $\lambda > 0$ and $N > 0$ as shape parameters.

- `geom` **Geom** (help). The probability mass function for *geom* is:

  $$f(k) = (1-p)^{k-1} p$$

  for $k \geq 1$, $0 < p \leq 1$

  geom takes p as shape parameter, where p is the probability of a single success and $1-p$ is the probability of a single failure.

- `logser` **Logser** (help). The probability mass function for *logser* is:

  $$f(k) = -\frac{p^k}{k \log(1-p)}$$

  for $k \geq 1$, $0 < p < 1$

  logser takes p as shape parameter, where p is the probability of a single success and $1-p$ is the probability of a single failure.

- **`nbinom` Nbinom (help). Negative binomial distribution describes a sequence of i.i.d. Bernoulli** trials, repeated until a predefined, non-random number of successes occurs.

  The probability mass function of the number of failures for *nbinom* is:

  $$f(k) = \binom{k+n-1}{n-1} p^n (1-p)^k$$

for $k \geq 0$, $0 < p \leq 1$

nbinom takes n and p as shape parameters where n is the number of successes, p is the probability of a single success, and $1-p$ is the probability of a single failure.

Another common parameterization of the negative binomial distribution is in terms of the mean number of failures μ to achieve n successes. The mean μ is related to the probability of success as

$$p = \frac{n}{n+\mu}$$

The number of successes n may also be specified in terms of a "dispersion", "heterogeneity", or "aggregation" parameter α, which relates the mean μ to the variance σ^2, e.g. $\sigma^2 = \mu + \alpha\mu^2$. Regardless of the convention used for α,

$$p = \frac{\mu}{\sigma^2}$$
$$n = \frac{\mu^2}{\sigma^2 - \mu}$$

- `planck` **Planck** (help). The probability mass function for *planck* is:

  $$f(k) = (1 - \exp(-\lambda)) \exp(-\lambda k)$$

  for $k \geq 0$ and $\lambda > 0$.

  planck takes λ as shape parameter. The Planck distribution can be written as a geometric distribution (*geom*) with $p = 1 - \exp(-\lambda)$ shifted by `loc = -1`.

- `poisson` **Poisson** (help). The probability mass function for *poisson* is:

  $$f(k) = \exp(-\mu) \frac{\mu^k}{k!}$$

  for $k \geq 0$.

  poisson takes $\mu \geq 0$ as shape parameter. When $\mu = 0$, the `pmf` method returns `1.0` at quantile $k = 0$.

- `randint` **Randint** (help). The probability mass function for *randint* is:

  $$f(k) = \frac{1}{\texttt{high} - \texttt{low}}$$

  for $k \in \{\texttt{low}, \dots, \texttt{high} - 1\}$.

  randint takes `low` and `high` as shape parameters.

5.1.10 Appendix: `scipy.stats` Continuous Random Variables

The information below was extracted from the scipy help for continuous distributions. The basic list can be created by introspection—wonderful Python!

```
In [1]: import scipy.stats as ss

In [2]: import pandas as pd

In [3]: ans = []
```

(continues on next page)

(continued from previous page)

```
In [4]: for k in dir(ss):
   ...:     ob = getattr(ss, k)
   ...:     if str(type(ob)).find('continuous_distns') > 0:
   ...:         try:
   ...:             fz = ob()
   ...:         except TypeError as e:
   ...:             ee = e
   ...:             ans.append([k, str(e), -1, ob.a, ob.b])
   ...:         else:
   ...:             ans.append([k, 'no args fine', 0, ob.a, ob.b])
   ...:

In [5]: df = pd.DataFrame(ans, columns=['dist', 'm', 'args', 'a', 'b'])

In [6]: for i in range(1,5):
   ...:     df.loc[df.m.str.find(f'{i} required')>=0, 'args'] = i
   ...:

In [7]: df = df.sort_values(['args', 'dist'])

In [8]: df['params'] = ''

In [9]: df.loc[df.args > 0, 'params'] = df.loc[df.args > 0, 'm'].str.split(':').
↪str[1]

In [10]: df = df.drop(columns='m')

In [11]: print(df.rename(columns={'dist': 'Distribution', 'args': 'Num. args',
   ....:         'a': 'Min range' , 'b': 'Max range', 'params': 'Parameters'}).\
   ....:         set_index('Distribution').to_string(float_format=lambda x: f'{x:.
↪4g}'))
   ....:
                    Num. args  Min range  Max range               Parameters
Distribution
anglit                      0    -0.7854     0.7854
arcsine                     0          0          1
cauchy                      0       -inf        inf
cosine                      0     -3.142      3.142
expon                       0          0        inf
gibrat                      0          0        inf
gumbel_l                    0       -inf        inf
gumbel_r                    0       -inf        inf
halfcauchy                  0          0        inf
halflogistic                0          0        inf
halfnorm                    0          0        inf
hypsecant                   0       -inf        inf
kstwobign                   0          0        inf
laplace                     0       -inf        inf
levy                        0          0        inf
levy_l                      0       -inf          0
logistic                    0       -inf        inf
maxwell                     0          0        inf
moyal                       0       -inf        inf
norm                        0       -inf        inf
rayleigh                    0          0        inf
semicircular                0         -1          1
uniform                     0          0          1
wald                        0          0        inf
alpha                       1          0        inf                      'a'
argus                       1          0          1                    'chi'
bradford                    1          0          1                      'c'
```

(continues on next page)

(continued from previous page)

```
chi                           1          0        inf                  'df'
chi2                          1          0        inf                  'df'
dgamma                        1       -inf        inf                   'a'
dweibull                      1       -inf        inf                   'c'
erlang                        1          0        inf                   'a'
exponnorm                     1       -inf        inf                   'K'
exponpow                      1          0        inf                   'b'
fatiguelife                   1          0        inf                   'c'
fisk                          1          0        inf                   'c'
foldcauchy                    1          0        inf                   'c'
foldnorm                      1          0        inf                   'c'
gamma                         1          0        inf                   'a'
genextreme                    1       -inf        inf                   'c'
genhalflogistic               1          0        inf                   'c'
genlogistic                   1       -inf        inf                   'c'
gennorm                       1       -inf        inf                'beta'
genpareto                     1          0        inf                   'c'
gompertz                      1          0        inf                   'c'
halfgennorm                   1          0        inf                'beta'
invgamma                      1          0        inf                   'a'
invgauss                      1          0        inf                  'mu'
invweibull                    1          0        inf                   'c'
kappa3                        1          0        inf                   'a'
ksone                         1          0          1                   'n'
kstwo                         1          0          1                   'n'
laplace_asymmetric            1       -inf        inf               'kappa'
loggamma                      1       -inf        inf                   'c'
loglaplace                    1          0        inf                   'c'
lognorm                       1          0        inf                   's'
lomax                         1          0        inf                   'c'
nakagami                      1          0        inf                  'nu'
pareto                        1          1        inf                   'b'
pearson3                      1       -inf        inf                'skew'
powerlaw                      1          0          1                   'a'
powernorm                     1       -inf        inf                   'c'
rdist                         1         -1          1                   'c'
recipinvgauss                 1          0        inf                  'mu'
rel_breitwigner               1          0        inf                 'rho'
rice                          1          0        inf                   'b'
skewcauchy                    1       -inf        inf                   'a'
skewnorm                      1       -inf        inf                   'a'
t                             1       -inf        inf                  'df'
triang                        1          0          1                   'c'
truncexpon                    1          0        inf                   'b'
tukeylambda                   1       -inf        inf                 'lam'
vonmises                      1       -inf        inf               'kappa'
vonmises_line                 1     -3.142      3.142               'kappa'
weibull_max                   1       -inf          0                   'c'
weibull_min                   1          0        inf                   'c'
wrapcauchy                    1          0      6.283                   'c'
beta                          2          0          1           'a' and 'b'
betaprime                     2          0        inf           'a' and 'b'
burr                          2          0        inf           'c' and 'd'
burr12                        2          0        inf           'c' and 'd'
crystalball                   2       -inf        inf        'beta' and 'm'
exponweib                     2          0        inf           'a' and 'c'
f                             2          0        inf       'dfn' and 'dfd'
gengamma                      2          0        inf           'a' and 'c'
geninvgauss                   2          0        inf           'p' and 'b'
jf_skew_t                     2       -inf        inf           'a' and 'b'
johnsonsb                     2          0          1           'a' and 'b'
```

(continues on next page)

(continued from previous page)

```
johnsonsu                    2       -inf        inf             'a' and 'b'
kappa4                       2       -inf        inf             'h' and 'k'
loguniform                   2       -inf        inf             'a' and 'b'
mielke                       2          0        inf             'k' and 's'
nct                          2       -inf        inf           'df' and 'nc'
ncx2                         2          0        inf           'df' and 'nc'
norminvgauss                 2       -inf        inf             'a' and 'b'
powerlognorm                 2          0        inf             'c' and 's'
reciprocal                   2       -inf        inf             'a' and 'b'
studentized_range            2          0        inf            'k' and 'df'
trapezoid                    2          0          1             'c' and 'd'
trapz                        2          0          1             'c' and 'd'
truncnorm                    2       -inf        inf             'a' and 'b'
truncpareto                  2          1        inf             'b' and 'c'
genexpon                     3          0        inf        'a', 'b', and 'c'
genhyperbolic                3       -inf        inf        'p', 'a', and 'b'
ncf                          3          0        inf   'dfn', 'dfd', and 'nc'
truncweibull_min             3       -inf        inf        'c', 'a', and 'b'
gausshyper                   4          0          1   'a', 'b', 'c', and 'z'
```

- `alpha` **Alpha** (help). The probability density function for *alpha* is:

$$f(x,a) = \frac{1}{x^2\Phi(a)\sqrt{2\pi}} * \exp(-\frac{1}{2}(a - 1/x)^2)$$

  where Φ is the normal CDF, $x > 0$, and $a > 0$.

  alpha takes `a` as a shape parameter.

- `anglit` **Anglit** (help). The probability density function for *anglit* is:

$$f(x) = \sin(2x + \pi/2) = \cos(2x)$$

  for $-\pi/4 \le x \le \pi/4$.

- `arcsine` **Arcsine** (help). The probability density function for *arcsine* is:

$$f(x) = \frac{1}{\pi\sqrt{x(1-x)}}$$

  for $0 < x < 1$.

- `argus` **Argus** (help). The probability density function for *argus* is:

$$f(x,\chi) = \frac{\chi^3}{\sqrt{2\pi}\Psi(\chi)} x\sqrt{1-x^2}\exp(-\chi^2(1-x^2)/2)$$

  for $0 < x < 1$ and $\chi > 0$, where

$$\Psi(\chi) = \Phi(\chi) - \chi\phi(\chi) - 1/2$$

  with Φ and ϕ being the CDF and PDF of a standard normal distribution, respectively.

  argus takes χ as shape a parameter.

- `beta` **Beta** (help). The probability density function for *beta* is:

$$f(x, a, b) = \frac{\Gamma(a+b)x^{a-1}(1-x)^{b-1}}{\Gamma(a)\Gamma(b)}$$

for $0 <= x <= 1$, $a > 0$, $b > 0$, where Γ is the gamma function (*scipy.special.gamma*).

beta takes a and b as shape parameters.

- `betaprime` **Beta Prime** (help). The probability density function for *betaprime* is:

$$f(x, a, b) = \frac{x^{a-1}(1+x)^{-a-b}}{\beta(a, b)}$$

for $x >= 0$, $a > 0$, $b > 0$, where $\beta(a, b)$ is the beta function (see *scipy.special.beta*).

betaprime takes `a` and `b` as shape parameters.

- `bradford` **Bradford** (help). The probability density function for *bradford* is:

$$f(x, c) = \frac{c}{\log(1+c)(1+cx)}$$

for $0 <= x <= 1$ and $c > 0$.

bradford takes `c` as a shape parameter for c.

- `burr` **Burr (Type III)** (help). The probability density function for *burr* is:

$$f(x, c, d) = cdx^{-c-1}/(1+x^{-c})^{d+1}$$

for $x >= 0$ and $c, d > 0$.

burr takes c and d as shape parameters.

This is the PDF corresponding to the third CDF given in Burr's list; specifically, it is equation (11) in Burr's paper. The distribution is also commonly referred to as the Dagum distribution. If the parameter $c < 1$ then the mean of the distribution does not exist and if $c < 2$ the variance does not exist. The PDF is finite at the left endpoint $x = 0$ if $c * d >= 1$.

- `burr12` **Burr (Type XII)** (help). The probability density function for *burr* is:

$$f(x, c, d) = cdx^{c-1}/(1+x^{c})^{d+1}$$

for $x >= 0$ and $c, d > 0$.

burr12 takes `c` and `d` as shape parameters for c and d.

This is the PDF corresponding to the twelfth CDF given in Burr's list; specifically, it is equation (20) in Burr's paper.

- `cauchy` **Cauchy** (help). The probability density function for *cauchy* is

$$f(x) = \frac{1}{\pi(1+x^2)}$$

for a real number x.

- `chi` **Chi** (help). The probability density function for *chi* is:

$$f(x,k) = \frac{1}{2^{k/2-1}\Gamma\left(k/2\right)} x^{k-1} \exp\left(-x^2/2\right)$$

for $x >= 0$ and $k > 0$ (degrees of freedom, denoted `df` in the implementation). Γ is the gamma function (*scipy.special.gamma*).

Special cases of *chi* are:

- `chi(1, loc, scale)` is equivalent to *halfnorm*
- `chi(2, 0, scale)` is equivalent to *rayleigh*
- `chi(3, 0, scale)` is equivalent to *maxwell*

chi takes `df` as a shape parameter.

- `chi2` **Chi-squared** (help). The probability density function for *chi2* is:

$$f(x,k) = \frac{1}{2^{k/2}\Gamma\left(k/2\right)} x^{k/2-1} \exp\left(-x/2\right)$$

for $x > 0$ and $k > 0$ (degrees of freedom, denoted `df` in the implementation).

chi2 takes `df` as a shape parameter.

The chi-squared distribution is a special case of the gamma distribution, with gamma parameters `a = df/2`, `loc = 0` and `scale = 2`.

- `cosine` **Cosine** (help). The cosine distribution is an approximation to the normal distribution. The probability density function for *cosine* is:

$$f(x) = \frac{1}{2\pi}(1+\cos(x))$$

for $-\pi \leq x \leq \pi$.

- `crystalball` **Crystalball** (help). The probability density function for *crystalball* is:

$$f(x,\beta,m) = \begin{cases} N\exp(-x^2/2), & \text{for } x > -\beta \\ NA(B-x)^{-m} & \text{for } x \leq -\beta \end{cases}$$

where $A = (m/|\beta|)^m \exp(-\beta^2/2)$, $B = m/|\beta| - |\beta|$ and N is a normalisation constant.

crystalball takes $\beta > 0$ and $m > 1$ as shape parameters. β defines the point where the pdf changes from a power-law to a Gaussian distribution. m is the power of the power-law tail.

- `dgamma` **Double Gamma** (help). The probability density function for *dgamma* is:

$$f(x,a) = \frac{1}{2\Gamma(a)}|x|^{a-1}\exp(-|x|)$$

for a real number x and $a > 0$. Γ is the gamma function (*scipy.special.gamma*).

dgamma takes `a` as a shape parameter for a.

- `dweibull` **Double Weibull** (help). The probability density function for *dweibull* is given by

$$f(x,c) = c/2|x|^{c-1}\exp(-|x|^c)$$

for a real number x and $c > 0$.

dweibull takes `c` as a shape parameter for c.

- `erlang` **Erlang** (help). The Erlang distribution is a special case of the Gamma distribution, with the shape parameter *a* an integer. Note that this restriction is not enforced by *erlang*. It will, however, generate a warning the first time a non-integer value is used for the shape parameter.

  Refer to *gamma* for examples.

- `expon` **Exponential** (help). The probability density function for *expon* is:

  $$f(x) = \exp(-x)$$

  for $x \ge 0$.

- `exponnorm` **Exponentially Modified Normal** (help). The probability density function for *exponnorm* is:

  $$f(x, K) = \frac{1}{2K} \exp\left(\frac{1}{2K^2} - x/K\right) \operatorname{erfc}\left(-\frac{x - 1/K}{\sqrt{2}}\right)$$

  where x is a real number and $K > 0$.

  It can be thought of as the sum of a standard normal random variable and an independent exponentially distributed random variable with rate `1/K`.

- `exponweib` **Exponentiated Weibull** (help). The probability density function for *exponweib* is:

  $$f(x, a, c) = ac[1 - \exp(-x^c)]^{a-1} \exp(-x^c) x^{c-1}$$

  and its cumulative distribution function is:

  $$F(x, a, c) = [1 - \exp(-x^c)]^a$$

  for $x > 0$, $a > 0$, $c > 0$.

  exponweib takes a and c as shape parameters:

  – a is the exponentiation parameter, with the special case $a = 1$ corresponding to the (non-exponentiated) Weibull distribution *weibull_min*.

  – c is the shape parameter of the non-exponentiated Weibull law.

- `exponpow` **Exponential Power** (help). The probability density function for *exponpow* is:

  $$f(x, b) = bx^{b-1} \exp(1 + x^b - \exp(x^b))$$

  for $x \ge 0$, $b > 0$. Note that this is a different distribution from the exponential power distribution that is also known under the names "generalized normal" or "generalized Gaussian".

  exponpow takes `b` as a shape parameter for b.

- `f` **F (Snecdor F)** (help). The probability density function for *f* is:

  $$f(x, df_1, df_2) = \frac{df_2^{df_2/2} df_1^{df_1/2} x^{df_1/2-1}}{(df_2 + df_1 x)^{(df_1+df_2)/2} B(df_1/2, df_2/2)}$$

  for $x > 0$.

  f takes `dfn` and `dfd` as shape parameters.

- `fatiguelife` **Fatigue Life (Birnbaum-Saunders)** (help). The probability density function for *fatiguelife* is:

$$f(x, c) = \frac{x+1}{2c\sqrt{2\pi x^3}} \exp(-\frac{(x-1)^2}{2xc^2})$$

for $x >= 0$ and $c > 0$.

fatiguelife takes `c` as a shape parameter for c.

- `fisk` **Fisk** (help). The probability density function for *fisk* is:

  $$f(x, c) = cx^{-c-1}(1 + x^{-c})^{-2}$$

  for $x >= 0$ and $c > 0$.

  fisk takes `c` as a shape parameter for c.

  fisk is a special case of *burr* or *burr12* with `d=1`.

- `foldcauchy` **Folded Cauchy** (help). The probability density function for *foldcauchy* is:

  $$f(x, c) = \frac{1}{\pi(1 + (x-c)^2)} + \frac{1}{\pi(1 + (x+c)^2)}$$

  for $x \geq 0$.

  foldcauchy takes `c` as a shape parameter for c.

- `foldnorm` **Folded Normal** (help). The probability density function for *foldnorm* is:

  $$f(x, c) = \sqrt{2/\pi} cosh(cx) \exp(-\frac{x^2 + c^2}{2})$$

  for $c \geq 0$.

  foldnorm takes `c` as a shape parameter for c.

- `genlogistic` **Generalized Logistic** (help). The probability density function for *genlogistic* is:

  $$f(x, c) = c\frac{\exp(-x)}{(1 + \exp(-x))^{c+1}}$$

  for $x >= 0$, $c > 0$.

  genlogistic takes `c` as a shape parameter for c.

- `gennorm` **Generalized normal** (help). The probability density function for *gennorm* is:

  $$f(x, \beta) = \frac{\beta}{2\Gamma(1/\beta)} \exp(-|x|^\beta)$$

  Γ is the gamma function (*scipy.special.gamma*).

  gennorm takes `beta` as a shape parameter for β. For $\beta = 1$, it is identical to a Laplace distribution. For $\beta = 2$, it is identical to a normal distribution (with `scale=1/sqrt(2)`).

- `genpareto` **Generalized Pareto** (help). The probability density function for *genpareto* is:

  $$f(x, c) = (1 + cx)^{-1-1/c}$$

defined for $x \geq 0$ if $c \geq 0$, and for $0 \leq x \leq -1/c$ if $c < 0$.

genpareto takes `c` as a shape parameter for c.

For $c = 0$, *genpareto* reduces to the exponential distribution, *expon*:

$$f(x, 0) = \exp(-x)$$

For $c = -1$, *genpareto* is uniform on `[0, 1]`:

$$f(x, -1) = 1$$

- `genexpon` **Generalized Exponential** (help). The probability density function for *genexpon* is:

  $$f(x, a, b, c) = (a + b(1 - \exp(-cx))) \exp(-ax - bx + \frac{b}{c}(1 - \exp(-cx)))$$

  for $x \geq 0$, $a, b, c > 0$.

  genexpon takes a, b and c as shape parameters.

- `genextreme` **Generalized Extreme Value** (help). For $c = 0$, *genextreme* is equal to *gumbel_r*. The probability density function for *genextreme* is:

  $$f(x, c) = \begin{cases} \exp(-\exp(-x)) \exp(-x) & \text{for } c = 0 \\ \exp(-(1 - cx)^{1/c})(1 - cx)^{1/c-1} & \text{for } x \leq 1/c, c > 0 \end{cases}$$

  Note that several sources and software packages use the opposite convention for the sign of the shape parameter c.

  genextreme takes `c` as a shape parameter for c.

- `gausshyper` **Gauss Hypergeometric** (help). The probability density function for *gausshyper* is:

  $$f(x, a, b, c, z) = Cx^{a-1}(1 - x)^{b-1}(1 + zx)^{-c}$$

  for $0 \leq x \leq 1$, $a > 0$, $b > 0$, $z > -1$, and $C = \frac{1}{B(a,b)F[2,1](c,a;a+b;-z)}$. $F[2, 1]$ is the Gauss hypergeometric function *scipy.special.hyp2f1*.

  gausshyper takes a, b, c and z as shape parameters.

- `gamma` **Gamma** (help). The probability density function for *gamma* is:

  $$f(x, a) = \frac{x^{a-1}e^{-x}}{\Gamma(a)}$$

  for $x \geq 0$, $a > 0$. Here $\Gamma(a)$ refers to the gamma function.

  gamma takes `a` as a shape parameter for a.

  When a is an integer, *gamma* reduces to the Erlang distribution, and when $a = 1$ to the exponential distribution.

  Gamma distributions are sometimes parameterized with two variables, with a probability density function of:

  $$f(x, \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1}e^{-\beta x}}{\Gamma(\alpha)}$$

  Note that this parameterization is equivalent to the above, with `scale = 1 / beta`.

- `gengamma` **Generalized gamma** (help). The probability density function for *gengamma* is:

  $$f(x, a, c) = \frac{|c| x^{ca-1} \exp(-x^c)}{\Gamma(a)}$$

  for $x \geq 0$, $a > 0$, and $c \neq 0$. Γ is the gamma function (*scipy.special.gamma*).

  gengamma takes a and c as shape parameters.

- `genhalflogistic` **Generalized Half Logistic** (help). The probability density function for *genhalflogistic* is:

  $$f(x, c) = \frac{2(1 - cx)^{1/(c-1)}}{[1 + (1 - cx)^{1/c}]^2}$$

  for $0 \leq x \leq 1/c$, and $c > 0$.

  genhalflogistic takes `c` as a shape parameter for c.

- `genhyperbolic` **Generalized Hyperbolic** (help). The probability density function for *genhyperbolic* is:

  $$f(x, p, a, b) = \frac{(a^2 - b^2)^{p/2}}{\sqrt{2\pi} a^{p-0.5} K_p\Big(\sqrt{a^2 - b^2}\Big)} e^{bx} \times \frac{K_{p-1/2}(a\sqrt{1 + x^2})}{(\sqrt{1 + x^2})^{1/2-p}}$$

  for $x, p \in (-\infty; \infty)$, $|b| < a$ if $p \geq 0$, $|b| \leq a$ if $p < 0$. $K_p(.)$ denotes the modified Bessel function of the second kind and order p (*scipy.special.kn*)

  genhyperbolic takes `p` as a tail parameter, `a` as a shape parameter, `b` as a skewness parameter.

- `geninvgauss` **Generalized Inverse Gaussian** (help). The probability density function for *geninvgauss* is:

  $$f(x, p, b) = x^{p-1} \exp(-b(x + 1/x)/2)/(2K_p(b))$$

  where *x > 0*, and the parameters *p, b* satisfy *b > 0*. K_p is the modified Bessel function of second kind of order *p* (*scipy.special.kv*).

- `gilbrat` **Gilbrat** (help). The probability density function for *gilbrat* is:

  $$f(x) = \frac{1}{x\sqrt{2\pi}} \exp(-\frac{1}{2}(\log(x))^2)$$

  gilbrat is a special case of *lognorm* with `s=1`.

- `gompertz` **Gompertz (Truncated Gumbel)** (help). The probability density function for *gompertz* is:

  $$f(x, c) = c \exp(x) \exp(-c(e^x - 1))$$

  for $x \geq 0$, $c > 0$.

  gompertz takes `c` as a shape parameter for c.

- `gumbel_r` (help). The probability density function for *gumbel_r* is:

  $$f(x) = \exp(-(x + e^{-x}))$$

  The Gumbel distribution is sometimes referred to as a type I Fisher-Tippett distribution. It is also related to the extreme value distribution, log-Weibull and Gompertz distributions.

- `gumbel_l` (help). The probability density function for *gumbel_l* is:

  $$f(x) = \exp(x - e^x)$$

  The Gumbel distribution is sometimes referred to as a type I Fisher-Tippett distribution. It is also related to the extreme value distribution, log-Weibull and Gompertz distributions.

- `halfcauchy` **Half Cauchy** (help). The probability density function for *halfcauchy* is:

  $$f(x) = \frac{2}{\pi(1 + x^2)}$$

  for $x \geq 0$.

- `halflogistic` **Half Logistic** (help). The probability density function for *halflogistic* is:

  $$f(x) = \frac{2e^{-x}}{(1 + e^{-x})^2} = \frac{1}{2}\text{sech}(x/2)^2$$

  for $x \geq 0$.

- `halfnorm` **Half Normal** (help). The probability density function for *halfnorm* is:

  $$f(x) = \sqrt{2/\pi}\exp(-x^2/2)$$

  for $x >= 0$.

  halfnorm is a special case of *chi* with `df=1`.

- `halfgennorm` **Generalized Half Normal** (help). The probability density function for *halfgennorm* is:

  $$f(x, \beta) = \frac{\beta}{\Gamma(1/\beta)}\exp(-|x|^\beta)$$

  for $x > 0$. Γ is the gamma function (*scipy.special.gamma*).

  gennorm takes `beta` as a shape parameter for β. For $\beta = 1$, it is identical to an exponential distribution. For $\beta = 2$, it is identical to a half normal distribution (with `scale=1/sqrt(2)`).

- `hypsecant` **Hyperbolic Secant** (help). The probability density function for *hypsecant* is:

  $$f(x) = \frac{1}{\pi}\text{sech}(x)$$

  for a real number x.

- `invgamma` **Inverse Gamma** (help). The probability density function for *invgamma* is:

  $$f(x, a) = \frac{x^{-a-1}}{\Gamma(a)}\exp(-\frac{1}{x})$$

  for $x >= 0$, $a > 0$. Γ is the gamma function (*scipy.special.gamma*).

  invgamma takes `a` as a shape parameter for a.

  invgamma is a special case of *gengamma* with `c=-1`, and it is a different parameterization of the scaled inverse chi-squared distribution. Specifically, if the scaled inverse chi-squared distribution is parameterized with degrees of freedom ν and scaling parameter τ^2, then it can be modeled using *invgamma* with `a=` $\nu/2$ and `scale=` $\nu\tau^2/2$.

- `invgauss` **Inverse Gaussian** (help). The probability density function for *invgauss* is:

  $$f(x, \mu) = \frac{1}{\sqrt{2\pi x^3}} \exp(-\frac{(x-\mu)^2}{2x\mu^2})$$

  for $x >= 0$ and $\mu > 0$.

  invgauss takes `mu` as a shape parameter for μ.

- `invweibull` **Inverse Weibull** (help). The probability density function for *invweibull* is:

  $$f(x, c) = cx^{-c-1} \exp(-x^{-c})$$

  for $x > 0$, $c > 0$.

  invweibull takes `c` as a shape parameter for c.

- `johnsonsb` **Johnson SB** (help). The probability density function for *johnsonsb* is:

  $$f(x, a, b) = \frac{b}{x(1-x)} \phi(a + b \log \frac{x}{1-x})$$

  where x, a, and b are real scalars; $b > 0$ and $x \in [0, 1]$. ϕ is the pdf of the normal distribution.

  johnsonsb takes a and b as shape parameters.

- `johnsonsu` **Johnson SU** (help). The probability density function for *johnsonsu* is:

  $$f(x, a, b) = \frac{b}{\sqrt{x^2 + 1}} \phi(a + b \log(x + \sqrt{x^2 + 1}))$$

  where x, a, and b are real scalars; $b > 0$. ϕ is the pdf of the normal distribution.

  johnsonsu takes a and b as shape parameters.

- `kappa4` **Kappa 4 parameter** (help). The probability density function for kappa4 is:

  $$f(x, h, k) = (1 - kx)^{1/k-1}(1 - h(1 - kx)^{1/k})^{1/h-1}$$

  if h and k are not equal to 0.

  If h or k are zero then the pdf can be simplified:

  h = 0 and k != 0:

  ```
  kappa4.pdf(x, h, k) = (1.0 - k*x)**(1.0/k - 1.0)*
                        exp(-(1.0 - k*x)**(1.0/k))
  ```

  h != 0 and k = 0:

  ```
  kappa4.pdf(x, h, k) = exp(-x)*(1.0 - h*exp(-x))**(1.0/h - 1.0)
  ```

  h = 0 and k = 0:

  ```
  kappa4.pdf(x, h, k) = exp(-x)*exp(-exp(-x))
  ```

kappa4 takes h and k as shape parameters.

The kappa4 distribution returns other distributions when certain h and k values are used.

h	k=0.0	k=1.0	-inf<=k<=inf
-1.0	Logistic logistic(x)		Generalized Logistic(1)
0.0	Gumbel gumbel_r(x)	Reverse Exponential(2)	Generalized Extreme Value genextreme(x, k)
1.0	Exponential expon(x)	Uniform uniform(x)	Generalized Pareto genpareto(x, -k)

- `kappa3` **Kappa 3 parameter** (help). The probability density function for *kappa3* is:

$$f(x, a) = a(a + x^a)^{-(a+1)/a}$$

  for $x > 0$ and $a > 0$.

  kappa3 takes `a` as a shape parameter for a.

- `ksone` **Distribution of Kolmogorov-Smirnov one-sided test statistic** (help). D_n^+ and D_n^- are given by

$$D_n^+ = \text{sup}_x (F_n(x) - F(x)),$$
$$D_n^- = \text{sup}_x (F(x) - F_n(x)),$$

  where F is a continuous CDF and F_n is an empirical CDF. *ksone* describes the distribution under the null hypothesis of the KS test that the empirical CDF corresponds to n i.i.d. random variates with CDF F.

- `kstwo` **Distribution of Kolmogorov-Smirnov two-sided test statistic** (help). D_n is given by

$$D_n = \text{sup}_x |F_n(x) - F(x)|$$

  where F is a (continuous) CDF and F_n is an empirical CDF. *kstwo* describes the distribution under the null hypothesis of the KS test that the empirical CDF corresponds to n i.i.d. random variates with CDF F.

- `kstwobign` **Limiting Distribution of scaled Kolmogorov-Smirnov two-sided test statistic.** (help). $\sqrt{n}D_n$ is given by

$$D_n = \text{sup}_x |F_n(x) - F(x)|$$

  where F is a continuous CDF and F_n is an empirical CDF. *kstwobign* describes the asymptotic distribution (i.e. the limit of $\sqrt{n}D_n$) under the null hypothesis of the KS test that the empirical CDF corresponds to i.i.d. random variates with CDF F.

- `laplace` **Laplace** (help). The probability density function for *laplace* is

$$f(x) = \frac{1}{2} \exp(-|x|)$$

  for a real number x.

- `laplace_asymmetric` (help). The probability density function for *laplace_asymmetric* is

$$\begin{aligned} f(x, \kappa) &= \frac{1}{\kappa+\kappa^{-1}} \exp(-x\kappa), \quad x \ge 0 \\ &= \frac{1}{\kappa+\kappa^{-1}} \exp(x/\kappa), \quad x < 0 \end{aligned}$$

for $-\infty < x < \infty$, $\kappa > 0$.

laplace_asymmetric takes `kappa` as a shape parameter for κ. For $\kappa = 1$, it is identical to a Laplace distribution.

- `levy` **Levy** (help). The probability density function for *levy* is:

  $$f(x) = \frac{1}{\sqrt{2\pi x^3}} \exp\left(-\frac{1}{2x}\right)$$

  for $x >= 0$.

  This is the same as the Levy-stable distribution with $a = 1/2$ and $b = 1$.

- `logistic` **Logistic** (help). The probability density function for *logistic* is:

  $$f(x) = \frac{\exp(-x)}{(1+\exp(-x))^2}$$

  logistic is a special case of *genlogistic* with `c=1`.

  Remark that the survival function (`logistic.sf`) is equal to the Fermi-Dirac distribution describing fermionic statistics.

- `loggamma` **Log-Gamma** (help). The probability density function for *loggamma* is:

  $$f(x, c) = \frac{\exp(cx - \exp(x))}{\Gamma(c)}$$

  for all $x, c > 0$. Here, Γ is the gamma function (*scipy.special.gamma*).

  loggamma takes `c` as a shape parameter for c.

- `loglaplace` **Log-Laplace (Log Double Exponential)** (help). The probability density function for *loglaplace* is:

  $$f(x, c) = \begin{cases} \frac{c}{2} x^{c-1} & \text{for } 0 < x < 1 \\ \frac{c}{2} x^{-c-1} & \text{for } x \ge 1 \end{cases}$$

  for $c > 0$.

  loglaplace takes `c` as a shape parameter for c.

- `lognorm` **Log-Normal** (help). The probability density function for *lognorm* is:

  $$f(x, s) = \frac{1}{sx\sqrt{2\pi}} \exp\left(-\frac{\log^2(x)}{2s^2}\right)$$

  for $x > 0$, $s > 0$.

  lognorm takes `s` as a shape parameter for s.

- `loguniform` **Log-Uniform** (help). The probability density function for this class is:

$$f(x, a, b) = \frac{1}{x \log(b/a)}$$

for $a \leq x \leq b$, $b > a > 0$. This class takes a and b as shape parameters.

- `lomax` **Lomax (Pareto of the second kind)** (help). The probability density function for *lomax* is:

  $$f(x, c) = \frac{c}{(1+x)^{c+1}}$$

  for $x \geq 0$, $c > 0$.

  lomax takes `c` as a shape parameter for c.

  lomax is a special case of *pareto* with `loc=-1.0`.

- `maxwell` **Maxwell** (help). A special case of a *chi* distribution, with `df=3`, `loc=0.0`, and given `scale = a`, where `a` is the parameter used in the Mathworld description.

  The probability density function for *maxwell* is:

  $$f(x) = \sqrt{2/\pi}x^2 \exp(-x^2/2)$$

  for $x >= 0$.

- `mielke` **Mielke's Beta-Kappa** (help). The probability density function for *mielke* is:

  $$f(x, k, s) = \frac{k x^{k-1}}{(1+x^s)^{1+k/s}}$$

  for $x > 0$ and $k, s > 0$. The distribution is sometimes called Dagum distribution. It was already defined in, called a Burr Type III distribution (*burr* with parameters `c=s` and `d=k/s`).

  mielke takes `k` and `s` as shape parameters.

- `moyal` **Moyal** (help). The probability density function for *moyal* is:

  $$f(x) = \exp(-(x + \exp(-x))/2)/\sqrt{2\pi}$$

  for a real number x.

- `nakagami` **Nakagami** (help). The probability density function for *nakagami* is:

  $$f(x, \nu) = \frac{2\nu^\nu}{\Gamma(\nu)} x^{2\nu-1} \exp(-\nu x^2)$$

  for $x >= 0$, $\nu > 0$.

  nakagami takes `nu` as a shape parameter for ν.

- `ncx2` **Non-central chi-squared** (help). The probability density function for *ncx2* is:

  $$f(x, k, \lambda) = \frac{1}{2} \exp(-(\lambda + x)/2) (x/\lambda)^{(k-2)/4} I_{(k-2)/2}(\sqrt{\lambda x})$$

  for $x >= 0$ and $k, \lambda > 0$. k specifies the degrees of freedom (denoted `df` in the implementation) and λ is the non-centrality parameter (denoted `nc` in the implementation). I_ν denotes the modified Bessel function of first order of degree ν (*scipy.special.iv*).

  ncx2 takes `df` and `nc` as shape parameters.

- `ncf` **Non-central F** (help). The probability density function for *ncf* is:

  $$f(x, n_1, n_2, \lambda) = \exp\left(\frac{\lambda}{2} + \lambda n_1 \frac{x}{2(n_1 x + n_2)}\right) n_1^{n_1/2} n_2^{n_2/2} x^{n_1/2-1}$$
  $$(n_2 + n_1 x)^{-(n_1+n_2)/2} \gamma(n_1/2)\gamma(1 + n_2/2)$$
  $$\frac{L_{n_2/2}^{\frac{n_1}{2}-1}\left(-\lambda n_1 \frac{x}{2(n_1 x + n_2)}\right)}{B(n_1/2, n_2/2)\gamma\left(\frac{n_1+n_2}{2}\right)}$$

  for $n_1, n_2 > 0$, $\lambda \geq 0$. Here n_1 is the degrees of freedom in the numerator, n_2 the degrees of freedom in the denominator, λ the non-centrality parameter, γ is the logarithm of the Gamma function, L_n^k is a generalized Laguerre polynomial and B is the beta function.

  ncf takes `df1`, `df2` and `nc` as shape parameters. If `nc=0`, the distribution becomes equivalent to the Fisher distribution.

- `nct` **Non-central Student's T** (help). If Y is a standard normal random variable and V is an independent chi-square random variable (*chi2*) with k degrees of freedom, then

  $$X = \frac{Y + c}{\sqrt{V/k}}$$

  has a non-central Student's t distribution on the real line. The degrees of freedom parameter k (denoted `df` in the implementation) satisfies $k > 0$ and the noncentrality parameter c (denoted `nc` in the implementation) is a real number.

- `norm` **Normal (Gaussian)** (help). The probability density function for *norm* is:

  $$f(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}$$

  for a real number x.

- `norminvgauss` **Normal Inverse Gaussian** (help). The probability density function for *norminvgauss* is:

  $$f(x, a, b) = \frac{a\, K_1(a\sqrt{1 + x^2})}{\pi\sqrt{1 + x^2}} \exp(\sqrt{a^2 - b^2} + bx)$$

  where x is a real number, the parameter a is the tail heaviness and b is the asymmetry parameter satisfying $a > 0$ and $|b| <= a$. K_1 is the modified Bessel function of second kind (*scipy.special.k1*).

- `pareto` **Pareto** (help). The probability density function for *pareto* is:

  $$f(x, b) = \frac{b}{x^{b+1}}$$

  for $x \geq 1$, $b > 0$.

  pareto takes `b` as a shape parameter for b.

- `pearson3` **Pearson type III** (help). The probability density function for *pearson3* is:

  $$f(x, \kappa) = \frac{|\beta|}{\Gamma(\alpha)} (\beta(x - \zeta))^{\alpha-1} \exp(-\beta(x - \zeta))$$

where:

$$\beta = \frac{2}{\kappa}$$
$$\alpha = \beta^2 = \frac{4}{\kappa^2}$$
$$\zeta = -\frac{\alpha}{\beta} = -\beta$$

Γ is the gamma function (*scipy.special.gamma*). Pass the skew κ into *pearson3* as the shape parameter `skew`.

- `powerlaw` **Power-function** (help). The probability density function for *powerlaw* is:

  $$f(x, a) = ax^{a-1}$$

  for $0 \le x \le 1$, $a > 0$.

  powerlaw takes `a` as a shape parameter for a.

- `powerlognorm` **Power log normal** (help). The probability density function for *powerlognorm* is:

  $$f(x, c, s) = \frac{c}{xs}\phi(\log(x)/s)(\Phi(-\log(x)/s))^{c-1}$$

  where ϕ is the normal pdf, and Φ is the normal cdf, and $x > 0$, $s, c > 0$.

  powerlognorm takes c and s as shape parameters.

- `powernorm` **Power normal** (help). The probability density function for *powernorm* is:

  $$f(x, c) = c\phi(x)(\Phi(-x))^{c-1}$$

  where ϕ is the normal pdf, and Φ is the normal cdf, and $x >= 0$, $c > 0$.

  powernorm takes `c` as a shape parameter for c.

- `rdist` **R-distribution** (help). The probability density function for *rdist* is:

  $$f(x, c) = \frac{(1 - x^2)^{c/2-1}}{B(1/2, c/2)}$$

  for $-1 \le x \le 1$, $c > 0$. *rdist* is also called the symmetric beta distribution: if B has a *beta* distribution with parameters (c/2, c/2), then X = 2*B - 1 follows a R-distribution with parameter c.

  rdist takes `c` as a shape parameter for c.

  This distribution includes the following distribution kernels as special cases:

  ```
  c = 2:  uniform
  c = 3:  `semicircular`
  c = 4:  Epanechnikov (parabolic)
  c = 6:  quartic (biweight)
  c = 8:  triweight
  ```

- `rayleigh` **Rayleigh** (help). The probability density function for *rayleigh* is:

  $$f(x) = x\exp(-x^2/2)$$

  for $x \ge 0$.

  rayleigh is a special case of *chi* with `df=2`.

- `rice` **Rice** (help). The probability density function for *rice* is:

  $$f(x, b) = x \exp(-\frac{x^2 + b^2}{2}) I_0(xb)$$

  for $x >= 0$, $b > 0$. I_0 is the modified Bessel function of order zero (*scipy.special.i0*).

  rice takes `b` as a shape parameter for b.

- `recipinvgauss` **Reciprocal Inverse Gaussian** (help). The probability density function for *recipinvgauss* is:

  $$f(x, \mu) = \frac{1}{\sqrt{2\pi x}} \exp\left(\frac{-(1-\mu x)^2}{2\mu^2 x}\right)$$

  for $x \geq 0$.

  recipinvgauss takes `mu` as a shape parameter for μ.

- `semicircular` **Semicircular** (help). The probability density function for *semicircular* is:

  $$f(x) = \frac{2}{\pi} \sqrt{1 - x^2}$$

  for $-1 \leq x \leq 1$.

  The distribution is a special case of *rdist* with *c = 3*.

- `skewcauchy` **Skew Cauchy** (help). The probability density function for *skewcauchy* is:

  $$f(x) = \frac{1}{\pi \left(\frac{x^2}{(a\, \text{sign}(x) + 1)^2} + 1\right)}$$

  for a real number x and skewness parameter $-1 < a < 1$.

  When $a = 0$, the distribution reduces to the usual Cauchy distribution.

- `skewnorm` **Skew normal** (help). The pdf is:

  ```
  skewnorm.pdf(x, a) = 2 * norm.pdf(x) * norm.cdf(a*x)
  ```

  skewnorm takes a real number a as a skewness parameter. When `a = 0` the distribution is identical to a normal distribution (*norm*).

- `studentized_range` (help). The probability density function for *studentized_range* is:

  $$f(x; k, \nu) = \frac{k(k-1)\nu^{\nu/2}}{\Gamma(\nu/2) 2^{\nu/2-1}} \int_0^\infty \int_{-\infty}^\infty s^\nu e^{-\nu s^2/2} \phi(z) \phi(sx + z) [\Phi(sx + z) - \Phi(z)]^{k-2} \,dz \,ds$$

  for $x ≥ 0$, $k > 1$, and $\nu > 0$.

  studentized_range takes `k` for k and `df` for ν as shape parameters.

  When ν exceeds 100,000, an asymptotic approximation (infinite degrees of freedom) is used to compute the cumulative distribution function.

- `t` **Student's T** (help). The probability density function for *t* is:

$$f(x, \nu) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1+x^2/\nu)^{-(\nu+1)/2}$$

where x is a real number and the degrees of freedom parameter ν (denoted `df` in the implementation) satisfies $\nu > 0$. Γ is the gamma function (*scipy.special.gamma*).

- `trapezoid` **Trapezoidal** (help). The trapezoidal distribution can be represented with an up-sloping line from `loc` to `(loc + c*scale)`, then constant to `(loc + d*scale)` and then downsloping from `(loc + d*scale)` to `(loc+scale)`. This defines the trapezoid base from `loc` to `(loc+scale)` and the flat top from `c` to `d` proportional to the position along the base with `0 <= c <= d <= 1`. When `c=d`, this is equivalent to *triang* with the same values for *loc*, *scale* and *c*.

  trapezoid takes c and d as shape parameters.

- `triang` **Triangular** (help). The triangular distribution can be represented with an up-sloping line from `loc` to `(loc + c*scale)` and then downsloping for `(loc + c*scale)` to `(loc + scale)`.

  triang takes `c` as a shape parameter for c.

- `truncexpon` **Truncated Exponential** (help). The probability density function for *truncexpon* is:

$$f(x, b) = \frac{\exp(-x)}{1 - \exp(-b)}$$

  for $0 <= x <= b$.

  truncexpon takes `b` as a shape parameter for b.

- `truncnorm` **Truncated Normal** (help). The standard form of this distribution is a standard normal truncated to the range [a, b] — notice that a and b are defined over the domain of the standard normal. To convert clip values for a specific mean and standard deviation, use:

```
a, b = (myclip_a - my_mean) / my_std, (myclip_b - my_mean) / my_std
```

  truncnorm takes a and b as shape parameters.

- `tukeylambda` **Tukey-Lambda** (help). A flexible distribution, able to represent and interpolate between the following distributions:

  - Cauchy ($lambda = -1$)
  - logistic ($lambda = 0$)
  - approx Normal ($lambda = 0.14$)
  - uniform from -1 to 1 ($lambda = 1$)

  tukeylambda takes a real number $lambda$ (denoted `lam` in the implementation) as a shape parameter.

- `uniform` **Uniform** (help). a uniform continuous random variable

- `vonmises` **Von-Mises (Circular)** (help). The probability density function for *vonmises* and *vonmises_line* is:

$$f(x, \kappa) = \frac{\exp(\kappa \cos(x))}{2\pi I_0(\kappa)}$$

  for $-\pi \le x \le \pi$, $\kappa > 0$. I_0 is the modified Bessel function of order zero (*scipy.special.i0*).

  vonmises is a circular distribution which does not restrict the distribution to a fixed interval. Currently, there is no circular distribution framework in scipy. The `cdf` is implemented such that `cdf(x + 2*np.pi) == cdf(x) + 1`.

vonmises_line is the same distribution, defined on $[-\pi, \pi]$ on the real line. This is a regular (i.e. non-circular) distribution.

vonmises and *vonmises_line* take `kappa` as a shape parameter.

- `vonmises_line` (help). The probability density function for *vonmises* and *vonmises_line* is:

$$f(x, \kappa) = \frac{\exp(\kappa \cos(x))}{2\pi I_0(\kappa)}$$

  for $-\pi \le x \le \pi$, $\kappa > 0$. I_0 is the modified Bessel function of order zero (*scipy.special.i0*).

  vonmises is a circular distribution which does not restrict the distribution to a fixed interval. Currently, there is no circular distribution framework in scipy. The `cdf` is implemented such that `cdf(x + 2*np.pi) == cdf(x) + 1`.

  vonmises_line is the same distribution, defined on $[-\pi, \pi]$ on the real line. This is a regular (i.e. non-circular) distribution.

  vonmises and *vonmises_line* take `kappa` as a shape parameter.

- `wald` **Wald** (help). The probability density function for *wald* is:

$$f(x) = \frac{1}{\sqrt{2\pi x^3}} \exp(-\frac{(x-1)^2}{2x})$$

  for $x >= 0$.

  wald is a special case of *invgauss* with `mu=1`.

- `weibull_min` (help). The probability density function for *weibull_min* is:

$$f(x, c) = c x^{c-1} \exp(-x^c)$$

  for $x > 0$, $c > 0$.

  weibull_min takes `c` as a shape parameter for c. (named k in Wikipedia article and a in `numpy.random.weibull`). Special shape values are $c = 1$ and $c = 2$ where Weibull distribution reduces to the *expon* and *rayleigh* distributions respectively.

- `weibull_max` (help). The probability density function for *weibull_max* is:

$$f(x, c) = c(-x)^{c-1} \exp(-(-x)^c)$$

  for $x < 0$, $c > 0$.

  weibull_max takes `c` as a shape parameter for c.

- `wrapcauchy` **Wrapped Cauchy** (help). The probability density function for *wrapcauchy* is:

$$f(x, c) = \frac{1-c^2}{2\pi(1+c^2 - 2c\cos(x))}$$

  for $0 \le x \le 2\pi$, $0 < c < 1$.

  wrapcauchy takes `c` as a shape parameter for c.

5.2 Quantiles and Related Risk Measures

Objectives: Definition and calculation of quantiles and related risk measures.

Audience: Readers interested in quantiles, VaR, and TVaR risk measures.

Prerequisites: Risk measures, probability.

See also: *Insurance Probability*.

Contents:

5.2.1 Helpful References

- Klugman *et al.* [2019]
- Mildenhall and Major [2022], Chapter 4
- Hyndman and Fan [1996]

5.2.2 Quantiles

A quantile function is inverse to the distribution function $F(x) := \mathsf{Pr}(X \le x)$. For each $0 < p < 1$, it solves $F(x) = p$ for x, answering the question,

> which x has non-exceedance probability equal to p?

Or, said another way,

> which x has exceedance probability equal to $1 - p$?

When the distribution function is continuous and strictly increasing there is a unique such x. It is called the p-quantile, and is denoted $q(p)$. The resulting function $q(p) = F^{-1}(p)$ is called the quantile function; it satisfies $F(q(p)) = p$.

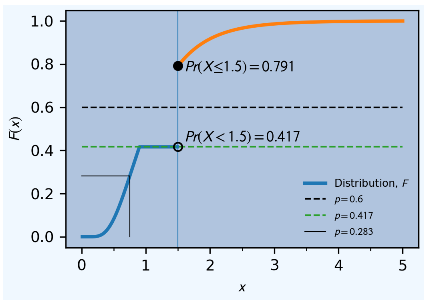
Two issues arise when defining quantiles.

1. The equation $F(x) = p$ may fail to have a unique solution when F is not strictly increasing. This can occur for any F. Is corresponds to a range of outcome values with probability zero.
2. When F is not continuous, the equation $F(x) = p$ may have no solution: F can jump from below p to above p. Simulation and catastrophe models, and all discrete random variables have discontinuous distributions.

Example.

Here's an example of the problems that can occur.

```
In [1]: from aggregate.extensions.pir_figures import fig_4_1

In [2]: fig = fig_4_1()
```



The distribution F has a flat spot between 0.9 and 1.5 at height $p = 0.417$. At $x = 1.5$ it jumps up to $p = 0.791$. The "inverse" to F at $p = 0.417$ could be any value between 0.9 and 1.5—illustrated by the lower green horizontal dashed line. The inverse at any value $0.417 < p < 0.791$ does not exist because there is no p so that $F(p) = 0.6$. However, any rational person looking at the graph would agree that the answer must be $x = 1.5$, where the black dashed line intersects the vertical line $x = 1.5$.

When F is not continuous and $F(x) = p$ has no solution because p lies is within a jump, we can still find an x so that

$$\mathsf{Pr}(X < x) \le p \le \mathsf{Pr}(X \le x).$$

$\mathsf{Pr}(X < x)$ equals the height of F at the bottom of the jump and $\mathsf{Pr}(X \le x)$ at the top. Turning this around, we can also say $\mathsf{Pr}(X \ge x) \ge 1-p \ge \mathsf{Pr}(X > x)$. At a p with no jump, $\mathsf{Pr}(X = x) = 0$, $\mathsf{Pr}(X < x) = p = \mathsf{Pr}(X \le x)$, and we have a well defined inverse, as the lower line at $p = 0.283$ illustrates.

The vertical segment at $x = 1.5$ between $p = 0.417$ and $p = 0.791$ is not strictly a part of F's graph, because a function must associate a *unique* value to each x in its domain. However, filling in the vertical segment makes it easier to locate inverse values by finding the graph's intersection with the horizontal line at p and is recommended in Rockafellar and Royset [2014]. Mentally, you should always fill in jumps in this way, treating the added segment as part of the graph.

Definition. Let X be a random variable with distribution function F and $0 < p < 1$. Any x satisfying

$$\mathsf{Pr}(X < x) \le p \le \mathsf{Pr}(X \le x)$$

is a p **quantile** of X. Any function $q(p)$ satisfying

$$\mathsf{Pr}(X < q(p)) \le p \le \mathsf{Pr}(X \le q(p))$$

for $0 \ < p < 1$ is a **quantile function** of X.

Exercise. What are the 0.1 and $1/6$ quantiles for the outcomes of the fair roll of a 6-sided die?

Solution. There are six outcomes $\{1, 2, 3, 4, 5, 6\}$ each with probability $1/6$. The distribution function jumps at each outcome.

1. For $p = 0.1$ we seek x so that $\mathsf{Pr}(X < x) \le 0.1 \le \mathsf{Pr}(X \le x)$. We know $0 = \mathsf{Pr}(X < 1) < \mathsf{Pr}(X \le 1) = 1/6$ and therefore $q(0.1) = 1$. It is good to rule out other possible values. If $x < 1$ then $\mathsf{Pr}(X \le x) = 0$ and if $x > 1$ then $\mathsf{Pr}(X < x) \ge 1/6$, showing neither alternative satisfies the definition of a quantile.
2. For $p = 1/6$ we seek x so that $\mathsf{Pr}(X < x) \le 1/6 \le \mathsf{Pr}(X \le x)$, which is satisfied by any $1 \le x \le 2$. If we pick $x = 1$ then $0 = \mathsf{Pr}(X < 1) < 1/6 = \mathsf{Pr}(X \le 1)$. If we pick $1 < x < 2$ then $\mathsf{Pr}(X < x) = 1/6 = \mathsf{Pr}(X \le x)$. If $x = 2$ then $\mathsf{Pr}(X < 2) = 1/6 < \mathsf{Pr}(X \le 2) = 1/3$.

Since the distribution and quantile functions are inverse, their graphs are reflections of one another in a 45-degree line through the origin. The distribution function is continuous from the right, hence the location of the probability masses indicated by the circles.

Define

- The **lower quantile** function $q^-(p) := \sup\ \{x \mid F(x) < p\} = \inf\ \{x \mid F(x) \ge p\}$, and

- The **upper quantile** function $q^+(p) := \sup\{x \mid F(x) \le p\} = \inf\{x \mid F(x) > p\}$.

The lower and upper quantiles both satisfy the requirements to be a quantile function. The lower quantile is left continuous. The upper quantile is right continuous. When the quantile is not unique, it lies between the lower and upper values.

5.2.3 Value at Risk

When a quantile is used as a risk measure it is called **Value at Risk (VaR)**: $\mathsf{VaR}_p(X) := q^-(p) = \inf\{x \mid F(x) \ge p\}$.

Thus l is $\mathsf{VaR}_p(X)$ if it is the smallest loss such that the probability $X \le l$ is $\ge p$. This is sometimes phrased: the smallest loss so that $X \le l$ with confidence at least p. *Smallest loss* allows for the case F is flat at p. *Probability* $\ge p$ allows for jumps in F.

VaR has several advantages. It is simple to explain, can be estimated robustly, and is always finite. It is widely used by regulators, rating agencies, and companies in their internal risk management. Its principal disadvantage is its failure to be subadditive.

5.2.4 The Failure of VaR to be Subadditive

It is easy to create simple discrete examples where VaR fails to be subadditive, for example:

Event	Prob	F	X_1	X_2	X
1	0.98	0.98	0	0	0
2	0.01	0.99	1000	100	1100
3	0.01	1.00	150	1100	1250

X_1 has 0.99 VaR 150 and X_2 has 0.99 VaR 100 but X has 0.99 VaR 1100.

More interesting, 0.7-VaR applied to the sum of two independent exponential distributions is not subadditive, but 0.95-VaR is.

```
In [3]: from aggregate import build, qd

In [4]: import pandas as pd

In [5]: p = build('port NotSA '
   ...:           'agg A dfreq [1] sev 1 * expon '
   ...:           'agg B dfreq [1] sev 1 * expon')
   ...:

In [6]: ans = p.var_dict(0.7)

In [7]: ans['sum'] = ans['A'] + ans['B']

In [8]: ans2 = p.var_dict(0.95)

In [9]: ans2['sum'] = ans2['A'] + ans2['B']

In [10]: pd.DataFrame([ans, ans2], index=pd.Index(['0.70', '0.95'], name='p'))
Out[10]:
          A      B  total    sum
p
0.70  1.204  1.204  2.439  2.408
0.95  2.996  2.996  4.744  5.992
```

The function `var_dict` returns the VaR of each unit in `p` and the total. The total VaR is greater than the sum of the parts. Subadditivity requires total VaR be less than or equal to the sum of the parts.

5.2.5 Tail VaR and Related Risk Measures

Tail value at risk (TVaR) is the conditional average of the worst $1-p$ outcomes. Let X be a loss random variable and $0 \le p < 1$. Then p-**Tail Value at Risk** is given by

$$\begin{aligned}\mathsf{TVaR}_p(X) &:= \frac{1}{1-p}\int_p^1 \mathsf{VaR}_s(X)\,ds \\ &= \frac{1}{1-p}\int_p^1 q^-(s)\,ds.\end{aligned}$$

In particular $\mathsf{TVaR}_0(X) = \mathsf{E}[X]$. When $p=1$, $\mathsf{TVaR}_1(X)$ is defined to be $\sup(X)$ if X is unbounded.

TVaR is defined in terms of q^-, that is, dual implicit events. The actual sample space on which X is defined is not used. Recall, $\mathsf{VaR}_p(X)$ refers to the lower quantile $q^-(p)$.

TVaR is a well behaved function of p. It is continuous, differentiable almost everywhere, and equal to the integral of its derivative (fundamental theorem of calculus). It takes every value between $\mathsf{E}[X]$ and $\sup X$. TVaR has a kink at jumps in F and is differentiable elsewhere.

Algorithm to Evaluate TVaR for a Discrete Distribution

Algorithm Input: X is a discrete random variable, taking N equally likely values $X_j \ge 0$, $j = 0, \dots, N-1$. Probability level p.

Follow these steps to determine $\mathsf{TVaR}_p(X)$.

Algorithm Steps

(1) **Sort** outcomes into ascending order $X_0 < \dots < X_{N-1}$.

(2) **Find** n so that $n \le pN < (n+1)$.

(3) **If** $n+1 = N$ **then** $\mathsf{TVaR}_p(X) := X_{N-1}$ is the largest observation, exit;

(4) **Else** $n < N-1$ and continue.

(5) **Compute** $T_1 := X_{n+1} + \dots + X_{N-1}$.

(6) **Compute** $T_2 := ((n+1) - pN)x_n$.

(7) **Compute** $\mathsf{TVaR}_p(X) := (1-p)^{-1}(T_1 + T_2)/N$.

These steps compute the average of the largest $N(1-p)$ observations. Step (6) adds a pro-rata portion of the $\lfloor N(1-p) \rfloor$ largest observation when $N(1-p)$ is not an integer. For instance, if $N = 71$ and $p = 0.95$, then $Np = 67.45$ and $n = 67$, giving $\mathsf{TVaR}_p = 20(0.55x_{67} + x_{68} + x_{69} + x_{70})/71$.

Example.

Let X be defined on a sample space with ten equally likely events and outcomes $0, 1, 1, 1, 2, 3, 4, 8, 12, 25$. Compute $\mathsf{TVaR}_p(X)$ for all p. Is it a piecewise linear function?

Solution. For $p \ge 0.9$, $q(p) = 25$ and $\mathsf{TVaR}_p(X) = 25$. For $0.8 \ge p < 0.9$

$$\begin{aligned}(1-p)\mathsf{TVaR}_p(X) &= \int_p^1 q^-(s)ds \\ &= \int_p^{0.9} q^-(s)ds + \int_{0.9}^1 q^-(s)ds \\ &= (0.9-p)\times 12 + (1-0.9)\times \mathsf{TVaR}_{0.9}(X),\end{aligned}$$

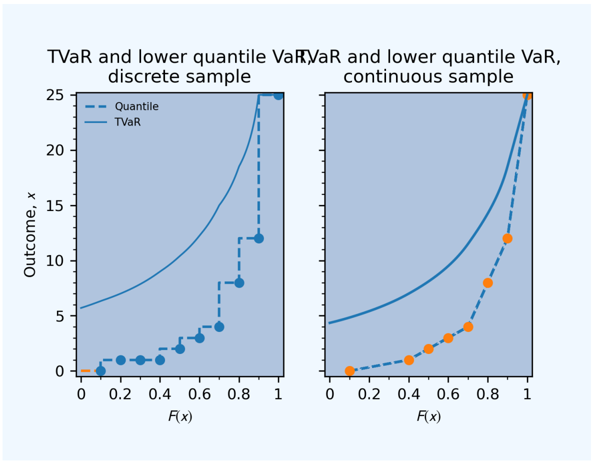
for $0.7 \ge p < 0.8$

$$(1-p)\mathsf{TVaR}_p(X) = (0.8-p)\times 8 + (1-0.8)\times \mathsf{TVaR}_{0.8}(X),$$

and so forth. The TVaR function is shown below. TVaR is not piecewise linear. For example, for $0.8 \le p < 0.9$, $\mathsf{TVaR}_p(X) = (12(0.9-p) + 2.5)/(1-p)$.

The default aggregate TVaR function ignores this slight non-linearity and just interpolates. To get a more exact answer use `kind='tail'`. The difference is illustrated on the left in the next figure.

```
In [11]: from aggregate.extensions.pir_figures import fig_4_8

In [12]: fig_4_8()
```



CTE, and WCE: Alternatives to TVaR

There are two other risk measures (confusingly) similar to TVaR.

1. Tail value at risk (TVaR) is the conditional average of the worst $1-p$ outcomes.
2. **Conditional tail expectation** (CTE) refers to the conditional expectation of X over $X \geq \mathsf{VaR}_p(X)$.
3. **Worst conditional expectation** (WCE) refers to the greatest expected value of X conditional on a set of probability $> 1-p$.

The formal definitions of CTE and WCE are as follows. Let X be a loss random variable and $0 \leq p < 1$.

- $\mathsf{CTE}_p(X) := \mathsf{E}[X \mid X \geq \mathsf{VaR}_p(X)]$ **(lower) conditional tail expectation** (TCE).
- The upper CTE equals $\mathsf{E}[X \mid X \geq q^+(p)]$.
- $\mathsf{WCE}_p(X) := \sup\{\mathsf{E}[X \mid A] \mid \mathsf{Pr}(A) > 1-p\}$ is the **worst conditional expectation**.

Like TVaR, CTE is defined in terms of quantiles, and the sample space on which X is defined is not used. In contrast, WCE works with the original sample space and relies on its events. Some actuarial papers refer to CTE as tail value at risk, e.g., Bodoff [2007].

For continuous random variables TVaR, CTE, and WCE are all equal, and they are easy to compute. The distinctions between them arise for discrete and mixed variables when p coincides with a mass point.

Expected Policyholder Deficit

The expected policyholder deficit **EPD** when a risk X is supported by assets a equals $\mathsf{E}[(X-a)^+]$, the unconditional excess loss cost. The insurer defaults on the EPD amount.

The **EPD ratio** is defined as the ratio of the EPD to expected losses. It gives the proportion of losses that are unpaid when X is supported by assets a.

Example.

We can use the EPD to define a tail risk measure that is analogous to VaR and TVaR. Define the **EPD risk measure** $\mathsf{E}PD_s(X)$ to be the amount of assets resulting in an EPD ratio of $0 < s < 1$, i.e., solving

$$\mathsf{E}[(X - \mathsf{E}PD_p(X))^+] = s\mathsf{E}[X].$$

The EPD risk measure is a stricter standard for smaller s. It accounts for the degree of default relative to promised payments, making it attractive to regulators. It is used to set risk based capital standards in Butsic [1994] and as a capital standard in Myers and Read Jr. [2001].

EPD is available in aggregate as the `epd` column in `density_df`.

5.3 Insurance Probability

Objectives: AEP and OEP points; theory of the insurance charge; adjusting severity distributions to achieve selected loss picks by layer in an excess of loss program; theory of the Tweedie distribution; when is severity relevant?

Audience: Actuaries at the Associate or Fellow level.

Prerequisites: Individual risk and retro rating; GLM modeling; Tweedie distributions.

See also: *Individual Risk Pricing*, *Reinsurance Pricing*, *The tweedie Keyword*.

Contents.

5.3.1 Helpful References

- Klugman *et al.* [2019]
- Panjer and Willmot [1992]
- Mildenhall and Major [2022]
- Woo [2002]

5.3.2 Occurrence and Aggregate Probable Maximal Loss

Probable maximal loss (PML)

Probable maximal loss or **PML** and the related **maximum foreseeable loss** (MFL) originated in fire underwriting in the early 1900s. The PML estimates the largest loss that a building is likely to suffer from a single fire if all critical protection systems function as expected. The MFL estimates the largest fire loss likely to occur if loss-suppression systems fail. For a large office building, the PML could be a total loss to 4 to 6 floors, and the MFL could be a total loss within four walls, assuming a single structure burns down. McGuinness [1969] discusses PMLs.

Today, PML is used to quantify potential catastrophe losses. Catastrophe risk is typically managed using reinsurance purchased on an occurrence basis and covering all losses from a single event. Therefore insurers are interested in the annual frequency of events greater than an attachment threshold, leading to the occurrence PML, now known as occurrence exceeding probabilities.

Occurrence Exceeding Probability (OEP)

To describe **occurrence PMLs**, we need to specify the stochastic model used to generate events. It is standard to use a homogeneous Poisson process, with a constant event intensity λ per year. The number of events in time t has a Poisson distribution with mean λt. If X is the severity distribution (size of loss conditional on an event) then the number of events per year above size x has Poisson distribution with mean $\lambda S(x)$. Therefore the probability of one or more events causing loss x or more is 1 minus the probability that a Poisson$(\lambda S(x))$ random variable equals zero, which equals $1-e^{-\lambda S(x)}$. The n **year occurrence PML**, $\mathsf{PML}_{n,\lambda}(X)=\mathsf{PML}_{n,\lambda}$, is the smallest loss x so that the probability of one or more events causing a loss of x or more in a year is at least $1/n$. It can be determined by solving $1-e^{-\lambda S(\mathsf{PML}_{n,\lambda})}=1/n$, giving

$$S(\mathsf{PML}_{n,\lambda}) = \frac{1}{\lambda}\log\left(\frac{n}{n-1}\right)$$
$$\implies \mathsf{PML}_{n,\lambda} = q_X\left(1-\frac{1}{\lambda}\log\left(\frac{n}{n-1}\right)\right)$$

(if $S(x)=s$ then $F(x)=1-s$ and $x=q_X(1-s)=\mathsf{VaR}_{1-s}(X)$). Thus, the occurrence PML is a quantile of severity at an adjusted probability level, where the adjustment depends on λ.

Converting to non-exceedance probabilities, if $p=1-1/n$ (close to 1) then $n/(n-1)=1/p$ and we obtain a relationship between the occurrence PML and severity VaR:

$$\mathsf{PML}_{n,\lambda} = q_X\left(1+\frac{\log(p)}{\lambda}\right) = \mathsf{VaR}_{1+\log(p)/\lambda}(X)$$

Catastrophe models output a sample of N loss events, each with an associated annual frequency λ_i and an expected loss x_i, $i=1,\dots,N$. Each event is assumed to have a Poisson occurrence frequency distribution. The associated severity distribution is concentrated on the set $\{x_1,\dots,x_N\}$ with $\mathsf{Pr}(X=x_i)=\lambda_i/\lambda$, where $\lambda=\sum_i\lambda_i$ is the expected annual event frequency. It is customary to fit or smooth X to get a continuous distribution, resulting in unique quantiles.

Return Periods

VaR points are often quoted by **return period**, such as a 100 or 250 year loss, rather than by probability level. By definition, the exceedance probability $\mathsf{Pr}(X>\mathsf{VaR}_p(X))$ of p-VaR is less than or equal to $1-p$, meaning at most a $1-p$ probability per year. If years are independent, then the average waiting time to an exceedance is at least $1/(1-p)$. (The waiting time has a geometric distribution, with parameter p. Let $q=1-p$. The average wait time is $q+2pq+3p^2q+\cdots=q(1+2p+3p^2+\cdots)=1/q$.)

Standard return periods and their probability representation are shown below.

VaR threshold	Exceedance probability	Return Period	Applications
p	$1-p$	$1/(1-p)$	
0.99	0.01	100 years	
0.995	0.005	200 years	Solvency 2
0.996	0.004	250 years	AM Best, S&P, RBC
0.999	0.001	1,000 years	

In a Poisson model, the waiting time between events with a frequency of λ has an exponential distribution with mean $1/\lambda$. Thus, an event with frequency 0.01 is often quoted as having a 100 year return period. Notice, however, the distinction between the chances of no events in a year and the waiting time until the next event. If λ is large, say 12 (one event per month on average), the chances of no events in a year equals $\exp(-12)=6.1\times10^{-6}$ is vs. a one-month return period. For small λ there is very little difference between the two since the probability of one or more events equals $1-\exp(-\lambda)\approx\lambda$.

To reiterate the definition above, when X represents aggregate annual losses, the statement $x=\mathsf{VaR}_{0.99}(X)$, $p=0.99$ means

- x is the smallest loss for which $X\le x$ with an annual probability of at least 0.99, or
- x is the smallest loss with an annual probability at most 0.01 of being exceeded.

Aggregate Exceeding Probability (AEP)

Severity VaR (quantile) and occurrence PML are distinct but related concepts. However, **aggregate PML** or **aggregate exceeding probability** is often used as a synonym for aggregate VaR, i.e., VaR of the aggregate loss distribution..

Let A equal the annual aggregate loss random variable. A has a compound Poisson distribution with expected annual frequency λ and severity random variable X. X is usually thick tailed. Then, as we explain shortly,

$$\mathsf{VaR}_p(A) \approx \mathsf{VaR}_{1-(1-p)/\lambda}(X).$$

This equation is a relationship between aggregate and severity VaRs.

We can sometimes estimate aggregate VaRs in terms of occurrence PMLs with no simulation. For large n and a thick tailed X occurrence PMLs and aggregate VaRs contain the same information—there is not more information in the aggregate, as is sometimes suggested. The approximation follows from the equation

$$\mathsf{Pr}(X_1 + \cdots + X_n > x) \to n\mathsf{Pr}(X > x) \text{ as } x \to \infty$$

for all n, which holds when X is sufficiently thick tailed. See Embrechts *et al.* [1997], Corollary 1.3.2 for the details.

5.3.3 Self-Insurance Plan Stop-Loss Insurance

Self-insurance plans often purchase per occurrence (specific) insurance, to limit the amount from any one loss that flows into the plan, and aggregate stop-loss insurance, to limit their aggregate liability over all occurrences in a year. Retro rating plans need to estimate the **insurance charge** for the aggregate cover. It is a function of the expected loss, the specific loss limit, and the aggregate retention. They sometimes also want to know the **insurance savings**, a credit for losses below a minimum. Tables tabulating insurance savings and charges are called Table L (California) or Table M (rest of the US). The two differ in the denominator: limited or unlimited losses.

Let X denote unlimited severity, N annual frequency, l the occurrence limit and a the aggregate retention of limited losses. The distribution of gross aggregate losses is given by

$$A_g := X_1 + \cdots + X_N.$$

Aggregate losses retained by the plan, reflecting the specific but not the aggregate insurance, are a function of l and $n := \mathsf{E}[N]$ the expected ground-up claim count, with distribution

$$A(n,l) := (X_1 \wedge l) + \cdots + (X_N \wedge l).$$

Aggregate limits are expressed in terms of the **entry ratio** r, which we define as the ratio

$$r = \frac{a}{\mathsf{E}[A(n,l)]}$$

of the aggregate limit to expected losses net of specific insurance. Therefore, the aggregate retention equals

$$a = r\mathsf{E}[A(n,l)] = rn\mathsf{E}[X_1 \wedge l].$$

The insurance charge

$$\begin{aligned}\phi(r) :&= \frac{\mathsf{E}\left[A(n,l)1_{A(n,l)>r\mathsf{E}[A(n,l)]}\right]}{\mathsf{E}[A(n,l)]} \\ &= \frac{\mathsf{E}\left[A(n,l) \mid A(n,l) > r\mathsf{E}[A(n,l)\right] S_{(n,l)}(r\mathsf{E}[A(n,l)])}{\mathsf{E}[A(n,l)]}\end{aligned}$$

where $S_{(n,l)}(\cdot)$ is the survival function of $A(n,l)$. The aggregate protection loss cost equals $\phi(r)\mathsf{E}[A(n,l)]$. The insurance savings equals

$$\begin{aligned}\psi(r) :&= \frac{\mathsf{E}\left[A(n,l)1_{A(n,l)\le r\mathsf{E}[A(n,l)]}\right]}{\mathsf{E}[A(n,l)]} \\ &= \frac{\mathsf{E}\left[A(n,l) \mid A(n,l) \le r\mathsf{E}[A(n,l)\right] F_{A(n,l)}(r\mathsf{E}[A(n,l)])}{\mathsf{E}[A(n,l)]}.\end{aligned}$$

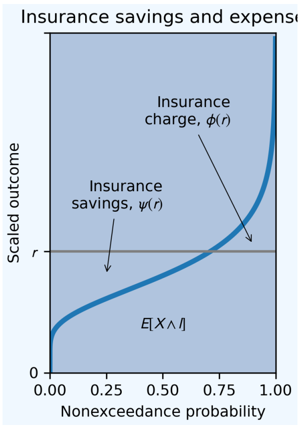
where $F_{(n,l)}(\cdot)$ is the cdf of $A(n,l)$.

With this notation, a retro program with maximum entry ratio r_1 and minimum r_0 has a net insurance charge (ignoring expenses and the loss conversion factor) equal to

$$(\phi(r_1) - \psi(r_0))n\mathsf{E}[X_1 \wedge l].$$

The charge and savings are illustrated below. Losses are scaled by expected (limited) losses in the figure and so the area under the blue curve equal 1. The graph is the Lee diagram, plotting x against $F(x)$.

```
In [1]: from aggregate.extensions.figures import savings_charge

In [2]: savings_charge();
```



The figure makes the put-call parity relationship, savings plus 1 equals entry plus charge obvious:

$$\psi(r) + 1 = r + \phi(r).$$

Remember r is the area under the horizontal line because the width of the plot equals 1. Taking $r = 1$ in put-call parity shows that $\psi(1) = \phi(1)$: at expected losses, the savings equals the charge.

5.3.4 Adjusting Layer Loss Picks

Reinsurance actuaries apply experience and exposure rating to excess of loss programs. Experience rating trends and develops layer losses to estimate loss costs. Exposure rating starts with a (ground-up) severity curve. In the US, these are often published by a rating agency (ISO, NCCI). It then applies a limit profit and uses difference of ILFs with a ground up loss ratio to estimate layer losses. The actuary then selects a loss cost by layer based on the two methods. When the selection is different from the exposure rate, the actuary no longer has a well-defined stochastic model for the business. In this section we show how to adjust the severity curve to match the selected loss picks by layer. The adjusted curve can then be used in a stochastic model that will replicate the layer loss selections.

Layer severity equals the integral of the survival function and layer expected losses equals layer frequency times severity. The easiest way to adjust a single layer is to scale the frequency. The simple approach fails when there are multiple layers because higher layer frequency impacts lower layers. We are led to adjust the survival function in each layer to hit the all selected layer loss picks. The method described next creates a legitimate, non-increasing survival function and retains its continuity properties whenever possible. It is easy to *select* inconsistent layer losses which produces negative probabilities or values greater than 1. When such inconsistencies occur the selections must be altered.

Here is the layer adjustment process. Adjustments to higher layers impact all lower layers because they change the probability of limit losses. The approach is to start from the top-most layer, figure its adjustment, and then take the impact of that adjustment into account on the next layer down, and so forth. The adjusted severity curve to maintain the shape of the curve and it continuity properties, conditional on a loss in each layer.

To make these ideas rigorous requires a surprising amount of notation. Define

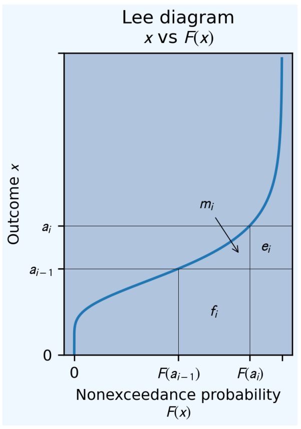
- Specify layer attachment points $0 = a_0 < a_1 < a_2 < \cdots < a_n$ and corresponding layer limits $y_i = a_i - a_{i-1}$ for $i = 1, 2, \dots, n$. The layers are l_i excess a_{i-1}.
- $l_i = \mathsf{LEV}(a_i) - \mathsf{LEV}(a_{i-1}) = \int_{a_{i-1}}^{a_i} S(x)dx = \mathsf{E}[(X - a_{i-1})^+ \wedge y_i]$ equals the unconditional expected layer loss (per ground-up claim).
- $p_i = \Pr(a_{i-1} < X \le a_i) = S(a_{i-1}) - S(a_i)$ equals the probability of a loss *in the layer*, excluding the mass at the limit.
- $e_i = y_i S(a_i)$ equals the part of l_i from full limit losses.
- $f_i = a_{i-1} p_i$
- $m_i = \int_{a_{i-1}}^{a_i} x dF(x) - f_i = \int_{a_{i-1}}^{a_i} (x - a_{i-1}) dF(x) = l_i - e_i$ equals the part of l_i from losses in the layer.
- t_i are selected unconditional expected losses by layer. $t_i = l_i$ results in no adjustment. t_i is computed by dividing the layer loss pick by the expected number of ground-up claims.

Integration by parts gives

$$\begin{aligned}\int_{a_{i-1}}^{a_i} S(x)dx &= xS(x)\,\Big|_{a_{i-1}}^{a_i} + \int_{a_{i-1}}^{a_i} x dF(x) \\ &= a_i S(a_i) + \int_{a_{i-1}}^{a_i} (x - a_{i-1}) dF(x) \\ &= e_i + m_i.\end{aligned}$$

These quantities are illustrated in the next figure.

```
In [1]: from aggregate.extensions.figures import adjusting_layer_losses

In [2]: adjusting_layer_losses();
```



There is no adjustment to S for $x \ge a_n$. In the top layer, adjust to $\tilde S(x) = S(a_n) + w_n(S(x) - S(a_n))$, so

$$\begin{aligned} t_n &= \int_{a_{n-1}}^{a_n} \tilde S(x)dx \\ &= S(a_n) y_n + w_n(l_n - e_n) \\ &= \omega_n y_n + w_n m_n \\ \implies w_n &= \frac{t_n - \omega_n y_n}{m_n},\end{aligned}$$

where $\omega_n = S(a_n)$. Set $\omega_i = \omega_{i+1} + w_{i+1} p_{i+1}$ and $\tilde S(x) = \omega_i + w_n(S(x) - S(a_n))$ in the ith layer. We can compute all the weights by proceeding down the tower:

$$\begin{aligned} t_i &= \int_{a_{i-1}}^{a_i} \tilde S(x)dx \\ &= \omega_i y_i + w_i(l_i - e_i) \\ \implies w_i &= \frac{t_i - \omega_i y_i}{m_i}.\end{aligned}$$

$\tilde{S}$ is continuous is S is because of the definition of ω at the layer boundaries. When $x = a_{i-1}$, $\tilde{S}(a_{i-1}) = \omega_i + w_i(S(a_{i-1}) - S(a_i)) = \omega_i + w_i p_i = \omega_{i=1}$.

The function `utilities.picks_work` computes the adjusted severity. In debug mode, it returns useful layer information. A severity can be adjusted on-the-fly by `Aggregate` using the `picks` keyword after the severity specification and before any occurrence reinsurance.

5.3.5 The Tweedie Distribution

The Tweedie distribution is a Poisson mixture of gammas. It is an exponential family distribution [Jørgensen, 1997]. Tweedie distributions are a suitable model for pure premiums and are used as unit distributions in GLMs [McCullagh and Nelder, 2019]. Tweedie distributions do not have a closed form density, but estimating the density is easy using `aggregate`.

The **Tweedie** family of distributions is a three-parameter exponential family. A variable $X \sim \mathrm{Tw}_p(\mu, \sigma^2)$ when $\mathsf{E}[X] = \mu$ and $\mathsf{Var}(X) = \sigma^2\mu^p$, $1 \le p \le 2$. p is a shape parameter and $\sigma^2 > 0$ is a scale parameter called the dispersion.

A Tweedie with $1 < p < 2$ is a compound Poisson distribution with gamma distributed severities. The limit when $p = 1$ is an over-dispersed Poisson and when $p = 2$ is a gamma. More generally: $\mathsf{Tw}_0(\mu, \sigma^2)$ is normal (μ, σ^2), $\mathsf{Tw}_1(\mu, \sigma^2)$ is over-dispersed Poisson $\sigma^2\mathsf{Po}(\mu/\sigma^2)$, and $\mathsf{Tw}_2(\mu, \sigma^2)$ is a gamma with CV σ.

Let $\mathsf{Ga}(\alpha, \beta)$ denote a gamma with shape α and scale β, with density $f(x; \alpha, \beta) = x^\alpha - e^{-x/\beta}/\beta^\alpha x\Gamma(\alpha)$. It has mean $\alpha\beta$, variance $\alpha\beta^2$, expected square $\alpha(\alpha+1)\beta$ and coefficient of variation $1/\sqrt{\alpha}$. We can define an alternative parameterization $\mathsf{Tw}^*(\lambda, \alpha, \beta) = \mathsf{CP}(\lambda, (Ga(\alpha, \beta))$ as a compound Poisson of gammas, with expected frequency λ.

The dictionary between the two parameterizations relies on the relation between the two shape parameters α and p given by

$$\alpha = \frac{2-p}{p-1}, \qquad p = \frac{2+\alpha}{1+\alpha}.$$

Starting from $\mathrm{Tw}_p(\mu, \sigma^2)$: $\lambda = \dfrac{\mu^{2-p}}{(2-p)\sigma^2}$ and $\beta = \dfrac{\mu^{1-p}}{(p-1)\sigma^2} = \mu/\lambda\alpha$

Starting from $\mathsf{Tw}^*(\lambda, \alpha, \beta)$: $\mu = \lambda\alpha\beta$ and $\sigma^2 = \lambda\alpha(\alpha+1)/(\beta^2\mu^p)$, by equating expressions for the variance.

It is easy to convert from the gamma mean m and CV ν to $\alpha = 1/\nu^2$ and $\beta = m/\alpha$. Remember, `scipy.stats` scale equals β.

Tweedie distributions are mixed: they have a probability mass of $p_0 = e^{-\lambda}$ at 0 and are continuous on $(0, \infty)$.

Jørgensen calls $\mathsf{Tw}(\lambda, \alpha, \beta)$ the **additive** form of the model because

$$\sum_i \mathsf{Tw}(\lambda_i, \alpha, \beta) = \mathsf{Tw}\left(\sum_i \lambda_i, \alpha, \beta\right).$$

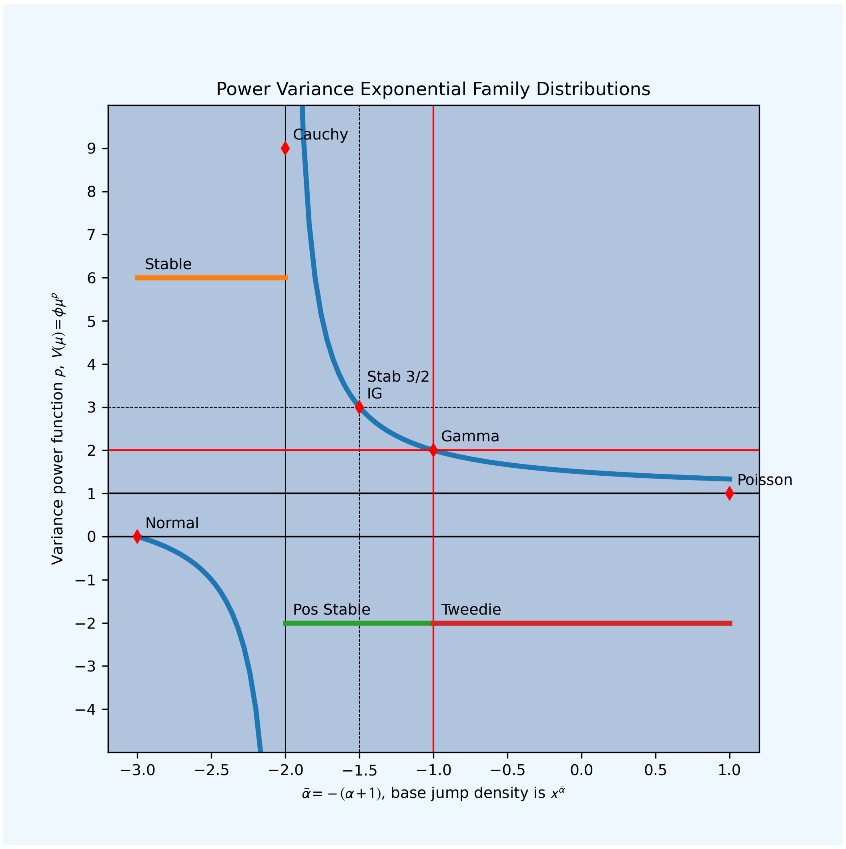
He calls $\mathsf{Tw}_p(\mu, \sigma)$ the **reproductive** exponential dispersion model. If $X_i \sim \mathsf{Tw}_p(\mu, \sigma/w_i)$ then

$$\frac{1}{w}\sum_i w_i X_i \sim \mathsf{Tw}_p\left(\mu, \frac{\sigma^2}{w}\right)$$

where $w = \sum_i w_i$. The weights w_i represents volume in cell i and X_i represents the pure premium. The sum on the left represents the total pure premium.

The next diagram shows how the Tweedie family fits within the broader power variance exponential family of distributions. See the blog post The Tweedie-Power Variance Function Family for more details.

```
In [1]: from aggregate.extensions.figures import power_variance_family

In [2]: power_variance_family()
```



5.3.6 Excess Frequency Distributions

Given a ground-up claim count distribution N, what is the distribution of the number of claims exceeding a certain threshold? We assume that severities are independent and identically distributed and that the probability of exceeding the threshold is q. Define an indicator variable I which takes value 0 if the claim is below the threshold and the value 1 if it exceeds the threshold. Thus $\mathsf{Pr}(I=0)=p=1-q$ and $\mathsf{Pr}(I=1)=q$. Let M_N be the moment generating function of N and N' is the number of claims in excess of the threshold. By definition we can express N' as an aggregate

$$N' = I_1 + \cdots + I_N.$$

Thus the moment generating function of N' is

$$\begin{aligned}M_{N'}(\zeta) &= M_N(\log(M_I(\zeta))) \\ &= M_N(\log(p+qe^\zeta))\end{aligned}$$

Using indicator variables I is called p-thinning by Grandell [1997].

Here are some examples.

Let N be Poisson with mean n. Then

$$M_{N'}(\zeta) = \exp(n(p+qe^\zeta-1)) = \exp(qn(e^\zeta-1))$$

so N' is also Poisson with mean qn—the simplest possible result.

Next let N be a G-mixed Poisson. Thus

$$\begin{aligned}M_{N'}(\zeta) &= M_N(\log(p+qe^\zeta)) \\ &= M_G(n(p+qe^\zeta-1)) \\ &= M_G(nq(e^\zeta-1)).\end{aligned}$$

Hence N' is also a G-mixed Poisson with lower underlying claim count nq in place of n.

In particular, if N has a negative binomial with parameters P and c (mean cP, $Q = 1 + P$, moment generating function $M_N(\zeta) = (Q - Pe^{\zeta})^{-1/c}$), then N' has parameters qP and c. If N has a Poisson-inverse Gaussian distribution with parameters μ and β, so

$$M_N(\zeta) = \exp\left(-\mu(\sqrt{1 + 2\beta(e^{\zeta} - 1)} - 1)\right),$$

then N is also Poisson inverse Gaussian with parameters μq and βq.

In all cases the variance of N' is lower than the variance of N and N' is closer to Poisson than N in the sense that the variance to mean ratio has decreased. For the general G-mixed Poisson the ratio of variance to mean decreases from $1 + cn$ to $1 + cqn$. As $q \to 0$ the variance to mean ratio approaches 1 and N' approaches a Poisson distribution. The fact that N' becomes Poisson is called the law of small numbers.

5.3.7 When Is Severity Irrelevant?

In some cases the actual form of the severity distribution is essentially irrelevant to the shape of the aggregate distribution. Consider an aggregate with a G-mixed Poisson frequency distribution. If the expected claim count n is large and if the severity is tame (roughly tame means bounded or has a log concave density; a policy with a limit has a tame severity; unlimited workers compensation or cat losses may not be tame) then particulars of the severity distribution diversify away in the aggregate. Moreover, the variability from the Poisson claim count component also diversifies away, and the shape of the aggregate distribution converges to the shape of the frequency mixing distribution G. Another way of saying the same thing is that the normalized distribution of aggregate losses (aggregate losses divided by expected aggregate losses) converges in distribution to G.

We can prove these assertions using moment generating functions. Let X_n be a sequence of random variables with distribution functions F_n and let X another random variable with distribution F. If $F_n(x) \to F(x)$ as $n \to \infty$ for every point of continuity of F then we say F_n converges weakly to F and that X_n converges in distribution to F.

Convergence in distribution is a relatively weak form of convergence. A stronger form is convergence in probability, which means for all $\epsilon > 0$ $\mathsf{Pr}(|X_n - X| > \epsilon) \to 0$ as $n \to \infty$. If X_n converges to X in probability then X_n also converges to X in distribution. The converse is false. For example, let $X_n = Y$ and $X = 1 - Y$ be binomial 0/1 random variables with $\mathsf{Pr}(Y = 1) = \mathsf{Pr}(X = 1) = 1/2$. Then X_n converges to X in distribution. However, since $\mathsf{Pr}(|X - Y| = 1) = 1$, X_n does not converge to X in probability.

It is a fact that X_n converges to X if the MGFs M_n of X_n converge to the MFG of M of X for all t: $M_n(t) \to M(t)$ as $n \to \infty$. See Feller [1971] for more details. We can now prove the following result.

Proposition. Let N be a G-mixed Poisson distribution with mean n, G with mean 1 and variance c, and let X be an independent severity with mean x and variance $x(1 + \gamma^2)$. Let $A = X_1 + \cdots + X_N$ and $a = nx$. Then A/a converges in distribution to G, so

$$\mathsf{Pr}(A/a < \alpha) \to \mathsf{Pr}(G < \alpha)$$

as $n \to \infty$. Hence

$$\sigma(A/a) = \sqrt{c + \frac{x(1 + \gamma^2)}{a}} \to \sqrt{c}.$$

Proof. We know

$$M_A(\zeta) = M_G(n(M_X(\zeta) - 1))$$

and so using Taylor's expansion we can write

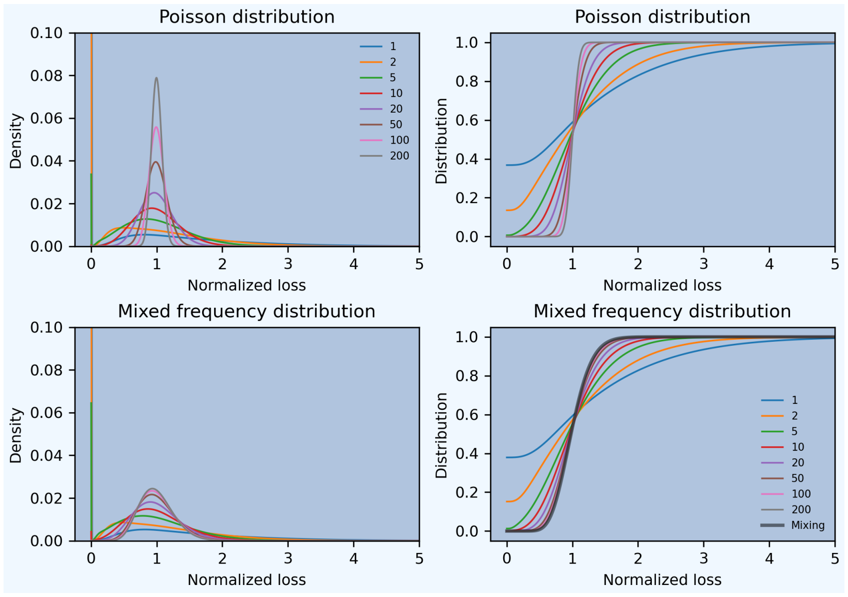
$$\begin{aligned}
\lim_{n\to\infty} M_{A/a}(\zeta) &= \lim_{n\to\infty} M_A(\zeta/a) \\
&= \lim_{n\to\infty} M_G(n(M_X(\zeta/nx) - 1)) \\
&= \lim_{n\to\infty} M_G(n(M_X'(0)\zeta/nx + R(\zeta/nx))) \\
&= \lim_{n\to\infty} M_G(\zeta + nR(\zeta/nx))) \\
&= M_G(\zeta)
\end{aligned}$$

for some remainder function $R(t) = O(t^2)$. Note that the assumptions on the mean and variance of X guarantee $M_X'(0) = x = \mathsf{E}[X]$ and that the remainder term in Taylor's expansion actually is $O(t^2)$. The second part is trivial.

The proposition implies that if the frequency distribution is actually a Poisson, so the mixing distribution $G = 1$ with probability 1, then the loss ratio distribution of a very large book will tend to the distribution concentrated at the expected, hence the expression that "with no parameter risk the process risk completely diversifies away."

The next figure illustrate the proposition, showing how aggregates change shape as expected counts increase.

```
In [1]: from aggregate.extensions import mixing_convergence

In [2]: mixing_convergence(0.25, 0.5)
```



On the top, $G = 1$ and the claim count is Poisson. Here the scaled distributions get more and more concentrated about the expected value (scaled to 1.0). Notice that the density peaks (left) are getting further apart as the claim count increases. The distribution (right) is converging to a Dirac delta step function at 1.

On the bottom, G has a gamma distribution with variance 0.0625 (asymptotic CV of 25%). The density peaks are getting closer, converging to the mixing gamma. The scaled aggregate distributions converge to G (thick line, right).

It is also interesting to compute the correlation between A and G. We have

$$\begin{aligned}\mathsf{cov}(A, G) &= \mathsf{E}[AG] - \mathsf{E}[A]\mathsf{E}[G] \\ &= \mathsf{E}\mathsf{E}[AG \mid G] - nx \\ &= \mathsf{E}[nxG^2] - nx \\ &= nxc,\end{aligned}$$

and therefore

$$\mathsf{corr}(A, G) = nxc/\sqrt{nx\gamma + n(1 + cn)}\sqrt{c} \to 1$$

as $n \to \infty$.

The proposition shows that in some situations severity is irrelevant to large books of business. However, it is easy to think of examples where severity is very important, even for large books of business. For example, severity becomes important in excess of loss reinsurance when it is not clear whether a loss distribution effectively exposes an excess layer. There, the difference in severity curves can amount to the difference between substantial loss exposure and none. The proposition does *not* say that any uncertainty surrounding the severity distribution diversifies away; it is only true when the severity distribution is known with certainty. As is often the case with risk management metrics, great care needs to be taken when applying general statements to particular situations!

5.4 Numerical Methods and FFT Convolution

Objectives: Describe the numerical distribution representation and FFT convolution algorithm that underlie all computations in `aggregate`.

Audience: User who wants a detailed understanding of the computational algorithms, potential errors, options, and parameters.

Prerequisites: Probability theory and aggregate distributions; complex numbers and matrix multiplication; numerical analysis, especially numerical integration; basics of Fourier transforms and series helpful.

See also: *User Guides*.

Contents:

5.4.1 Helpful References

Actuarial and operational risk books and papers

- Gerber [1982]
- Bühlmann [1984]
- Embrechts *et al.* [1993]
- Wang [1998]
- Grübel and Hermesmeier [1999]
- Mildenhall [2005]
- Schaller and Temnov [2008]
- Kaas *et al.* [2008]
- Embrechts and Frei [2009]
- Shevchenko [2010]

Books on probability covering characteristic functions, $t \mapsto \mathsf{E}[e^{itX}]$

- Loeve [1955]
- Feller [1971]
- Lukacs [1970]
- Billingsley [1986]
- Malliavin *et al.* [1995]
- McKean [2014]

Books on Fourier analysis and Fourier transforms, $t \mapsto \mathsf{E}[e^{-2\pi itX}]$, the same concept with slightly different notation. Malliavin is a sophisticated treatment of both Fourier analysis and probability.

- Stein and Weiss [1971]
- Stein and Shakarchi [2011]
- Strang [1986]

- Terras [2013]
- Körner [2022]

5.4.2 Overview

A Trilemma

Numerical analysts face a trilemma: they can pick two of fast, accurate, or flexible. Simulation is flexible, but trades speed against accuracy. Fewer simulations is faster but less accurate; many simulations improves accuracy at the cost of speed. `aggregate` delivers the third trilemma option using a fast Fourier transform (FFT) convolution algorithm to deliver speed and accuracy but with less flexibility than simulation.

The `aggregate` Convolution Algorithm

> **Complaint** It is the most natural thing in the world to decompose an oscillating electrical signal into sines and cosines of various frequencies via the Fourier transform - but probabilities, no. Basically, these are positive numbers adding up to 1, and what have sines and cosines to do with that? Indeed, in many applications done first by Fourier, a simpler, more understandable proof may emerge upon taking Fourier away. Still, the Fourier transform is a very effective, sometimes indispensable technical tool in much of our business here. [McKean, 2014]

This section describes the core convolution algorithm implemented in `aggregate`. It is an application where the caveat in McKean's complaint holds true. The use of Fourier transform methods is unnatural but very effective. As usual, let $A = X_1 + \cdots + X_N$ where severity X_i are iid and independent of the claim count N. We want to explicitly compute the distribution

$$\begin{aligned}\Pr(A < a) &= \sum_n \Pr(A < a \mid N = n)\Pr(N = n)\\ &= \sum_n \Pr(X^{*n} < a)\Pr(N = n).\end{aligned}$$

Here, X^{*n} denotes the sum of n independent copies of X. This problem is usually hard to solve analytically. For example, if X is lognormal then there is no closed form expression for the distribution of $X^{*2} \sim X_1 + X_2$. However, things are more promising in the Fourier domain. The characteristic function of A can be written, using independence, as

$$\begin{aligned}\phi_A(t) :&= \mathsf{E}[e^{itA}]\\ &= \mathsf{E}[\mathsf{E}[e^{itA} \mid N]]\\ &= \mathsf{E}[\mathsf{E}[e^{itX}]^N]\\ &= \mathscr{P}_N(\phi_X(t))\end{aligned}$$

where $\mathscr{P}_N(z) = \mathsf{E}[z^N]$ is the probability generating function. The pgfs of most common frequency distributions are know. For example, N is Poisson with mean λ then it is easy to see that $\mathscr{P}_N(z) = \exp(\lambda(z-1))$.

Note: The algorithm assumes the pgf can be written an explicit function. That, in turn, implies that frequency is thin tailed (it is not subexponential). This is generally not a problem because of how insurance events are defined. Losses from a catastrophe event, which can produce a large number of claims, are combined into a single occurrence. As a result, we usually model a small and thin tailed number of events with a thick tailed severity distribution. For example, there are approximately 1.75 US landfalling hurricanes per year, and the distribution of events since 1850 is well-fit by a (thin-tailed) Poisson distribution.

Knowing the characteristic function is useful because it can be inverted to determine the density of A. Subject to certain terms and conditions, described below, these arguments can be carried out in a finite discrete setting which is helpful for two reasons. First, it makes the problem tractable for a digital computer, and second, many A that arise in insurance problems that are discrete or mixed, because policy limits introduce mass points. For these reasons, we model a discrete approximation to A, see also num how agg reps a dist.

In a discrete approximation, the **function** $\phi_X(t)$ is replaced by a **vector sample** computed either by taking the discrete Fourier transform (DFT) of a discretized approximation to the distribution of X or by directly sampling ϕ_X. We usually do the former, computing the DFT using the Fast Fourier transform (FFT) algorithm. However, certain X, such as stable distributions, have known characteristic functions but no closed-form distribution or density function. In that case, a sample of the characteristic function can be used directly. The pgf is then applied element-by-element to the characteristic function sample vector, and the inverse FFT routine used to obtain a discrete approximation to the aggregate distribution. The exact rationale for this process are discussed in num ft convo. The errors it introduces are discussed there and in num error analysis.

In summary, the **FFT algorithm** is simply:

1. Discretize the severity cdf.
2. Apply the FFT to discrete severity.
3. Apply the frequency pgf to the FFT.
4. Apply the inverse FFT to create a discretized approximation to the aggregate distribution.

This algorithm has appeared numerous times in the literature, see *Related Actuarial Literature*. The details are laid out in num algo details. A plain Python implementation is presented in *Aggregate Algorithm in Detail*.

Strengths and Weaknesses

I've been using the FFT algorithm since Glenn Meyers explained it to me at a COTOR meeting around 1996, and I still find it miraculous. It is very fast and its speed is largely independent of the expected claim count—in contrast to simulations. The algorithm is also very accurate, both in absolute and relative terms. It is essentially exact in many cases and eight-plus digit precision is often easy to obtain. The algorithm works well in almost all situations and for many use-cases it is unbeatable, including computing:

- The distribution of aggregate losses from a portfolio with a complex limit and attachment profile, and a mixed severity.
- Ceded or net outcome distributions for an occurrence reinsurance program.
- Ceded or net outcome distributions for reinsurance contracts with variable features such as sliding commissions, swing rated programs, profit commissions, aggregate limits, see *Reinsurance Pricing*.
- The distribution of retained losses net of specific and aggregate insurance, as found in a risk-retention group, see *Individual Risk Pricing*, including exact computation of Table L and Table M charges in US worker compensation ratemaking.
- The distribution of the sum of independent distributions, e.g., aggregating units such as line of business, business unit, geographic unit etc.
- The distribution of the sum of dependent units, where the dependence structure is driven by *common frequency mixing variables*.

The algorithm is particularly well-suited to compute aggregates with low claim counts and a thick-tailed severity and where accuracy is important, such as catastrophe risk PML, AEP, and OEP points. Outcomes with low expected loss rates are hard to simulate accurately.

The FFT algorithm is not a panacea. On the downside, its mysterious Fourier-nature presents the user with a choice of trusting in magic or a steep learning curve to understand the theory. It relies on hard-to-select parameters and can fail spectacularly and without warning if they are not chosen judiciously. A big contribution of `aggregate` is to provide the user with sensible default parameters and a test of model validity, see num error analysis. It does not work well for a high mean, thick-tailed frequency combined with a thick-tailed severity distribution that has an intricate distribution—an unusual situation that stresses any numerical method. However, when either frequency or severity is thin-tailed, it excels. Finally, the `aggergate` implementation is limited to tracking one variable at a time. It cannot model joint distributions, such as ceded and net loss or derived quantities such as the total cession to a specific and aggregate cover, or the cession to an occurrence program with limited reinstatements. Both of these require a bivariate distribution. It *can* model the net position after specific and aggregate cessions, and ceded losses to an occurrence program with an aggregate limit. See num extensions for an approach to bivariate extensions.

Related Actuarial Literature

The earliest reference to Fourier transforms in actuarial science I have found is Heckman and Meyers [1983]. They used continuous Fourier transforms to compute aggregate distributions by numerically integrating the characteristic function. Their analysis includes severity and frequency uncertainty, that they call contagion and mixing.

Explicit use of the FFT appears first in [Bertram, 1983]. It has subsequently appeared in numerous places.

Bühlmann [1984] compares the FFT algorithm with Panjer recursion for compound Poisson distributions. It concludes that usually FFTs can be computed in fewer operations. Hürlimann [1986] obtains an error bound for stop-loss premium computed with FFTs.

Robertson [1992] considers a quasi-FFT algorithm, using discrete-continuous adjustments to reflect a piecewise linear as opposed to a fully discrete, distribution function. These greatly complicate the analysis for little tangible benefit. We recommend using a fully discrete distribution as explained in num how agg reps a dist.

Embrechts *et al.* [1993] describes the FFT algorithm and considers Richardson extrapolation to estimate the density.

Wang [1998] describes the FFT algorithm, using padding to control aliasing (wrapping) error. The first edition of Klugman *et al.* [2019], published in 1998, describes the algorithm, although it no longer appears in the fifth edition. Grübel and Hermesmeier [1999] describes the use of exponential tilting to reduce aliasing error and Grübel and Hermesmeier [2000] explains how to use Richardson extrapolation to improve density estimates. Exponential tilting is the same process used in GLM exponential families to adjust the mean, and it is also used in large deviation theory. Mildenhall [2005] describes the FFT algorithm.

Approximate inversion of the Fourier transform is also possible using FFTs. Menn and Rachev [2006] uses of FFTs to determine densities for distributions which have analytic MGFs but not densities, notably the class of stable distributions. This method is shown in num ft convo.

Kaas *et al.* [2008] section 3.6 presents the FFT algorithm in R.

Embrechts and Frei [2009] revisits Panjer recursion compared to the FFT algorithm. It also explores exponential tilting for aliasing error. It comments "Compared to the Panjer recursion, FFT has two main advantages: It works with arbitrary frequency distributions and it is much more efficient." It concludes:

> The Panjer recursion is arguably the most widely used method to "exactly" evaluate compound distributions. However, FFT is a viable alternative: It can be applied with arbitrary frequencies and offers a tremendous timing advantage for a large number of lattice points; moreover, the use of exponential tilting—which practically rules out any aliasing effects—facilitates applications (such as evaluation of the lower tail) that were thought to be an exclusive task for recursive procedures.

More recently, Papush *et al.* [2021], extending Papush *et al.* [2001], considers the best two parameter approximation to an frequency severity convolution. It shows that the gamma provides the best fit across a wide range of synthetic examples. However, all of their examples have a bounded (hence thin tailed) severity. A simple model:

```
agg 10 claims sev lognorm 2 poisson
```

is not best fit by a gamma.

Homer and Clark [2003] and Mildenhall [2005] describe the use of two-dimensional FFTs to model aggregates with bivariate frequency (for two different lines) and bivariate severity (net and ceded).

Other Applications

The FFT algorithm is applied to model operational risk in Schaller and Temnov [2008], Temnov and Warnung [2008], Luo and Shevchenko [2009], Luo and Shevchenko [2011], and Shevchenko [2010]. These applications mirror the actuarial approach, using either padding or exponential tilting (exponential window) to control aliasing error. They are interesting because they include modeling with a very high expected claim counts and a very thick tailed severity (no mean). See num truncation example.

In finance, FFTs are used in option pricing, Carr and Madan [1999]. These applications can use distributions derived from stable-α and Levy process families that have a closed for characteristic function but no analytic density. Duan *et al.* [2012] describe more recent innovations. FFTs are also used as a general purpose convolution routine, Černý [2004]

Wilson and Keich [2016] describes an interesting approach to accurate pairwise convolution that splits each component to limit the ratio of its most and least (non-zero) likely outcome. It provides helpful estimates for assessing relative error and determining when an FFT estimate can be trusted.

Conditional Expectations (Kappa)

The function $\kappa_i(x) := \mathsf{E}[X_i \mid X = x]$ is the basis for many of the computations in `Portfolio`. It can be computed using Fourier transforms because it is a convolution. There is no loss in generality assuming $X = X_1 + X_2$. For simplicity suppose (X_1, X_2) have a bivariate density f. Then

$$\begin{aligned}\mathsf{E}[X_1 \mid X = x] &= \int_0^x t\frac{f(t, x-t)}{f(x)}\,dt \\ &= \frac{1}{f(x)}\int_0^x tf_1(t)f_2(x-t)\,dt\end{aligned}$$

can be computed from the convolution of $tf_1(t)$ and f_2. The convolution can be computed using Fourier transforms in the usual way: transform, product, inverse transform. Using FFTs and relying on the discretized version of X_i, the algorithm becomes:

1. Compute the discrete approximation to $X_{\hat{i}}$, the sum of all X_j, $j \neq i$, identifying distributions with their discrete approximation vectors.
2. Compute FFTs of X_i and $X_{\hat{i}}$, with optional padding.
3. Take the elementwise product of the FFTs.
4. Apply the inverse FFT and unpad if necessary.

A variable with density $xf(x)/\mathsf{E}[X]$ is called the size-bias of X. Size-biased variables have lots of interesting applications, see Arratia *et al.* [2019].

The `aggregate` implementation computes $X_{\hat{i}}$ by dividing out the distribution of X_i from the overall sum (deconvolution), where that is possible, saving computing time.

5.4.3 Digital Representation of Distributions

> "We come now to reality. The truth is that the digital computer has totally defeated the analog computer. The input is a sequence of numbers and not a continuous function. The output is another sequence of numbers." [Strang, 1986]

How `aggregate` Represents a Distribution

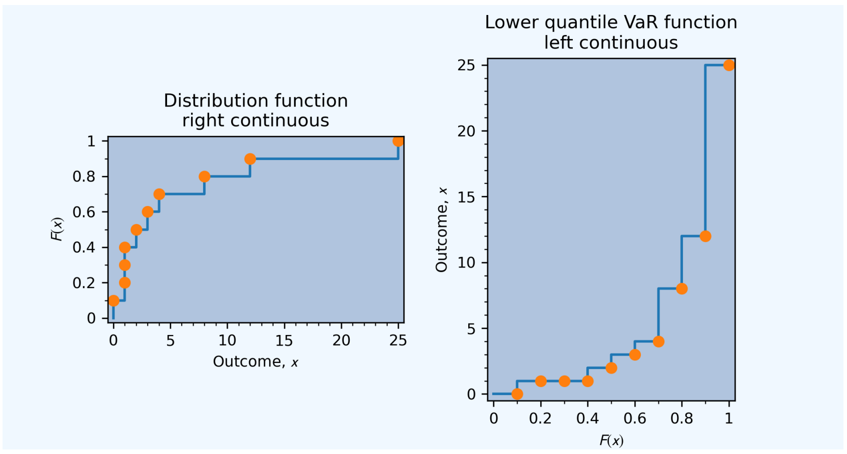
`aggregate` aims to deliver the speed and accuracy of parametric distributions to aggregate probability distributions and make them as easy to use as the lognormal. To achieve that, it needs a representation of the underlying distribution amenable to computation.

There is no analytic expression for the cdf of most aggregate distributions. For example, there is no closed form expression for the distribution of the sum of two lognormals [Milevsky and Posner, 1998]. Therefore we must use a numerical approximation to the exact cdf. There are two obvious ways to construct a numerical approximation to a cdf:

1. As a discrete (arithmetic, lattice) distribution supported on a discrete set of points.
2. As a continuous random variable with a piecewise linear distribution function.

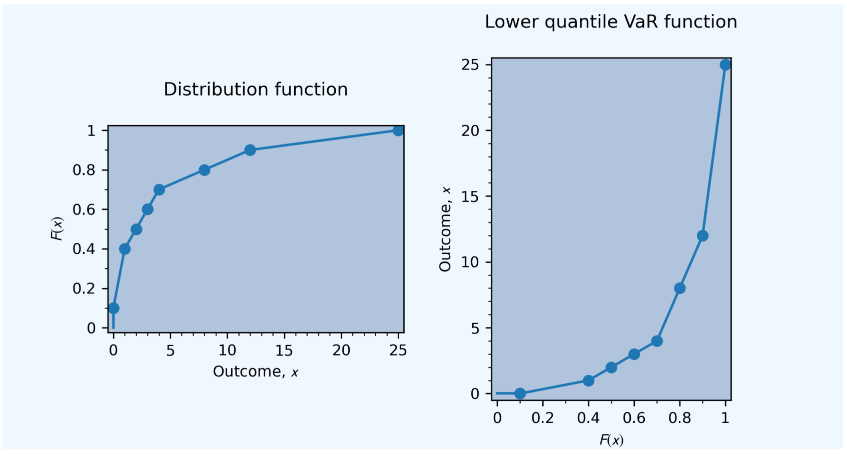
The next two figures illustrate of the two approaches. First, a discrete approximation, which results in a step-function, piecewise constant cdf, is shown left and the corresponding quantile function, right. The cdf is continuous from the right and the (lower) quantile function is continuous from the left. The distribution does not have a density function (pdf); it only has a probability mass function (pmf).

```
In [1]: from aggregate.extensions.pir_figures import fig_4_5, fig_4_6

In [2]: fig_4_5()
```



Second, a piecewise linear continuous approximation, which results in a step-function pdf (not shown).

```
In [3]: fig_4_6()
```



The second approach assumes the aggregate has a continuous distribution, which is often not the case. For example, the Tweedie and all other compound Poisson distributions are mixed (they have a mass at zero). An aggregate whose severity has a limit will have a mass at multiples of the limit caused by the non-zero probability of limit-only claims. When X is mixed it is impossible to distinguish the jump and continuous parts using a numerical approximation. The large jumps may be obvious but the small ones are not.

There are three other arguments in favor of discrete models. First, we live in a discrete world. Monetary amounts are multiples of a smallest unit: the penny, cent, yen, satoshi. Computers are inherently discrete. Second, probability theory is based on measure theory, which approximates distributions using simple functions that are piecewise constant. Third, the continuous model introduces unnecessary complexities in use, without any guaranteed gain in accuracy across all cases. See the complicated calculations in Robertson [1992], for example.

For all of these reasons, we use a discrete approximation. Further, we assume that the distribution is known at integer multiples of a fixed bandwidth or bucket size. This assumption is forced by the use of FFTs and has some undesirable consequences. Ideally, we would use a stratified approach, sampling more points where the distribution changes shape and using larger gaps to capture the tail. However, the computational efficiency of FFTs make this a good trade-off.

Based on the above considerations, saying we have **computed an aggregate** means that we have a discrete approximation to its distribution function concentrated on integer multiples of a fixed bucket size b. This specifies the approximation aggregate as

1. the value b and
2. a vector of probabilities $(p_0, p_1, \dots, p_{n-1})$

with the interpretation

$$\Pr(X = kb) = p_k.$$

All subsequent computations assume that the aggregate is approximated in this way. There are several important consequences.

- The cdf is a step function with a jump of size p_k at kb.
- The cdf is continuous from the right (it jumps up at kb).
- The cdf be computed from the cumulative sum of $(p_0, p_1, \dots, p_{n-1})$.
- The approximation has moments given by

$$\sum_k k^r p_i b.$$

- The limited expected value (the integral of the survival function), can be computed at the points kb as b times the cumulative sum of the survival function.
- The pdf, if it exists, can be approximated by p_i/b.

All of these calculations are more straightforward than assuming a piecewise linear cdf.

Sidebar: Continuous Discretization

It is possible to *approximate* the continuous cdf approach in `aggregate`. For example, the following code will reproduce the simple example in Section 4 of Robertson [1992]. Compare the output to his Table 4. Using `bs=1/200` approximates a continuous histogram. The use of a decimal bucket size is never recommended, but is used here to approximate Robertson's table values. We recommend against this approach. It is unnecessarily complicated and does not improve accuracy in any example we have encountered.

```
In [4]: from aggregate import build, qd

In [5]: s = build('agg Robertson '
   ...:           '5 claims '
   ...:           'sev chistogram xps [0 .2 .4 .6 .8 1] [.2 .2 .2 .2 .2] '
   ...:           'fixed'
   ...:           , bs=1/200, log2=12)
   ...:

In [6]: qd(s.density_df.loc[0:6:40, ['F']], max_rows=100,
   ...:    float_format=lambda x: f'{x:.10f}')
   ...:

                    F
loss
0.000    0.0000000000
200.000m 0.0000028505
400.000m 0.0000881354
600.000m 0.0006619547
800.000m 0.0027744084
1.000    0.0084395964
1.200    0.0209413662
1.400    0.0447838984
1.600    0.0847627931
1.800    0.1446976504
2.000    0.2261520701
2.200    0.3271821577
2.400    0.4417953515
```

(continues on next page)

(continued from previous page)

```
2.600    0.5611682516
2.800    0.6755414579
3.000    0.7761395705
3.200    0.8570626840
3.400    0.9164675600
3.600    0.9559897985
3.800    0.9794889995
4.000    0.9917687630
4.200    0.9973109416
4.400    0.9993650547
4.600    0.9999172021
4.800    0.9999974838
5.000    1.0000000000
5.200    1.0000000000
5.400    1.0000000000
5.600    1.0000000000
5.800    1.0000000000
6.000    1.0000000000
```

Discretizing the Severity Distribution

This section discusses ways to approximate a severity distribution with a discrete distribution. Severity distributions used by `aggregate` are supported on the non-negative real numbers; we allow a loss of zero, but not negative losses. However, the discretization process allows severity to be derived from a distribution supported on the whole real line—see the note below.

Let F be a distribution function and q the corresponding lower quantile function. It is convenient to be able to refer to a random variable with distribution F, so let $X = q(U)$ where $U(\omega) = \omega$ is the standard uniform variable on the sample space $\Omega = [0, 1]$. X has distribution F [Föllmer and Schied, 2016].

We want approximate F with a finite, purely discrete distribution supported at points $x_k = kb$, $k = 0, 1, \dots, m$, where b is called the **bucket size** or the **bandwidth**. Split this problem into two: first create an infinite discretization on $k = 0, 1, \dots$, and then truncate it.

The calculations described in this section are performed in `Aggregate.discretize()`.

Infinite Discretization

There are four common methods to create an infinite discretization.

1. The **rounding** method assigns probability to the k th bucket equal to

$$\begin{aligned}
p_k &= \Pr((k-1/2)b < X \le (k+1/2)b) \\
&= F((k+1/2)b) - F((k-1/2)b) \\
p_0 &= F(b/2).
\end{aligned}$$

2. The **forward** difference method assigns

$$\begin{aligned}
p_k &= \Pr(kb < X \le (k+1)b) \\
&= F((k+1)b) - F(kb) \\
p_0 &= F(b).
\end{aligned}$$

3. The **backward** difference method assigns

$$\begin{aligned}
p_k &= \Pr((k-1)b < X \le kb) \\
&= F(kb) - F((k-1)b) \\
p_0 &= F(0).
\end{aligned}$$

4. The **moment** difference method [Klugman *et al.*, 2019] assigns

$$p_k = \frac{2\mathsf{E}[X\wedge kb] - \mathsf{E}[X\wedge (k-1)b] - \mathsf{E}[X\wedge (k+1)b]}{b}$$
$$p_0 = 1 - \frac{\mathsf{E}[X\wedge b]}{b}.$$

   The moment difference ensures the discretized distribution has the same first moment as the original distribution. This method can be extended to match more moments, but the resulting weights are not guaranteed to be positive.

Note: Setting the first bucket to $F(b/2)$ for the rounding method (resp. $F(b)$, $F(0)$) allows the use of random variables with negative support. Any values ≤ 0 are included in the zero bucket. This behavior is useful because it allows the normal, Cauchy, and other similar distributions can be used as the basis for a severity.

Each of these methods produces a sequence $p_k \ge 0$ of probabilities that sum to 1 that can be interpreted as the pmf and distribution function $F_b^{(d)}$ of a discrete approximation random variable $X_b^{(d)}$

$$\Pr(X_b^{(d)} = kb) = p_k$$
$$F_b^{(d)}(kb) = \sum_{i\le k} p_i$$

where superscript $d = r,\ f,\ b,\ m$ describes the discretization method and subscript b the bucket size.

There is a disconnect between how the rounding method is defined and how it is interpreted. By definition, it corresponds to a distribution with jumps at $(k+1/2)b$, not kb. However, the approximation assumes the jumps are at kb to simplify and harmonize subsequent calculations across the three discretization methods.

It is clear that [Embrechts and Frei, 2009]

$$\begin{aligned}
F_b^{(b)} &\le F \le F_b^{(f)}\\
F_b^{(b)} &\le F_b^r \le F_b^{(f)}\\
X_b^{(b)} &\ge X \ge X_b^{(f)}\\
X_b^{(b)} &\ge X_b^r \ge X_b^{(f)}\\
X_b^{(b)} &\uparrow X \text{ as } b \downarrow 0\\
X_b^{(f)} &\downarrow X \text{ as } b \downarrow 0
\end{aligned}$$

X_b, X_r, and X_f converge weakly (in L^1) to X and the same holds for a compound distribution with severity X. These inequalities are illustrated in the example below.

Rounding Method Used by Default

`aggregate` uses the **rounding** method by default and offers the forward and backwards methods to compute explicit bounds on the distribution approximation if required. We found that the rounding method performs well across all examples we have run. These options are available in `update()` through the `sev_calc` argument, which can take the values `round`, `forwards`, and `backwards`. This decision is based in part on the following observations about the moment method in Embrechts and Frei [2009] (emphasis added):

> that both the forward/backward differences and the rounding method do not conserve any moments of the original distribution. In this light Gerber [1982] suggests a procedure that locally matches the first k moments. Practically interesting is only the case $k = 1$; for $k \ge 2$ the procedure is not well defined, potentially leading to negative probability mass on certain lattice points. The moment matching method is **much more involved than the rounding method** in terms of implementation; we need to calculate limited expected values. Apart from that, the **gain is rather modest**; moment matching only pays off for large bandwidths, and after all, **the rounding method is to be preferred**. This is further reinforced by the work of Grübel and Hermesmeier [1999]: if the severity distribution is absolutely continuous

with a sufficiently smooth density, the quantity $f_{b,j}/b$, an approximation for the compound density, can be quadratically extrapolated.

Klugman *et al.* [2019] report that Panjer and Lutek [1983] found two moments were usually sufficient and that adding a third moment requirement adds only marginally to the accuracy. Furthermore, they report that the **rounding method and the first-moment method had similar errors**, while the second-moment method provided significant improvement but at the cost of no longer guaranteeing that the resulting probabilities are **nonnegative**.

Approximating the Density

The pdf at kb can be approximated as p_k / b. This suggests another approach to discretization. Using the rounding method

$$\begin{aligned} p_k &= F((k+1/2)b) - F((k-1/2)b) \\ &= \int_{(k-1/2)b}^{(k+1)b} f(x)dx \\ &\approx f(kb)b. \end{aligned}$$

Therefore we could rescale the vector $(f(0), f(b), f(2b), \dots)$ to have sum 1. This method works well for continuous distributions, but does not apply for mixed ones, e.g., when a policy limit applies.

Discretization Example

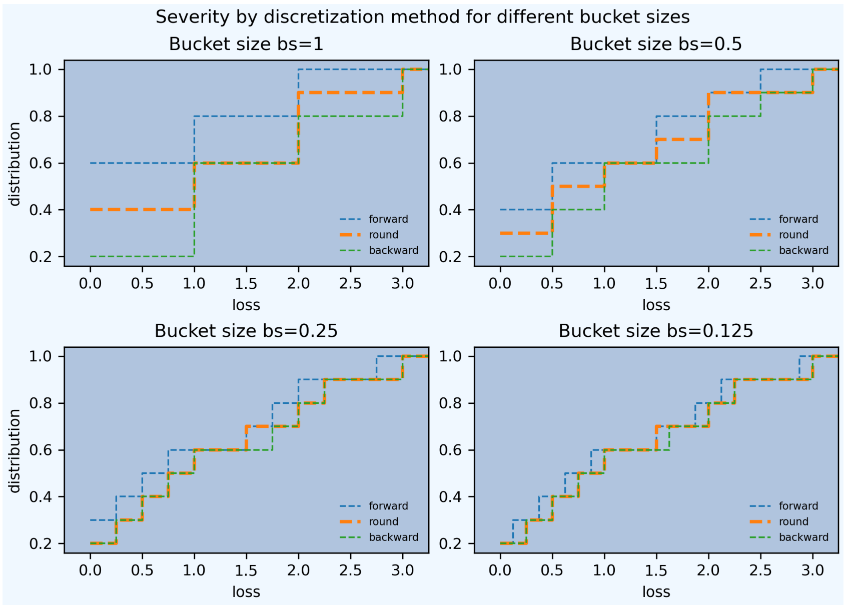
This example illustrates the impact of different discretization methods on the severity and aggregate distributions. The example uses a severity that can take negative values. `aggregate` treats any negative values as a mass at zero. This approach allows for the use of the normal and other distributions supported on the whole real line. The severity has finite support, so truncation is not an issue, and it is discrete so it is easy to check the calculations are correct. The severity is shown first, discretized using `bs=1, 1/2, 1/4, 1/8`. As expected, the rounding method (orange), lies between the forward (blue) and backwards (green) methods.

```
In [7]: from aggregate import build, qd

In [8]: import matplotlib.pyplot as plt

In [9]: from matplotlib import ticker

In [10]: dsev = [-1, 0, .25, .5, .75, 1, 1.5 + 1 / 16, 2, 2 + 1/4, 3]

In [11]: a01 = build(f'agg Num:01 1 claim dsev {dsev} fixed', update=False)

In [12]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.45 + 0.1),
   ....:     constrained_layout=True)
   ....:

In [13]: for bs, ax in zip([1, 1/2, 1/4, 1/8], axs.flat):
   ....:     for k in ['forward', 'round', 'backward']:
   ....:         a01.update(log2=10, bs=bs, sev_calc=k)
   ....:         a01.density_df.p_total.cumsum().\
   ....:             plot(xlim=[-.25, 3.25], lw=2 if  k=='round' else 1,
   ....:             drawstyle='steps-post', ls='--', label=k, ax=ax)
   ....:         ax.legend(loc='lower right')
   ....:         ax.set(title=f'Bucket size bs={bs}')
   ....:

In [14]: axs[0,0].set(ylabel='distribution');

In [15]: axs[1,0].set(ylabel='distribution');
```

(continues on next page)

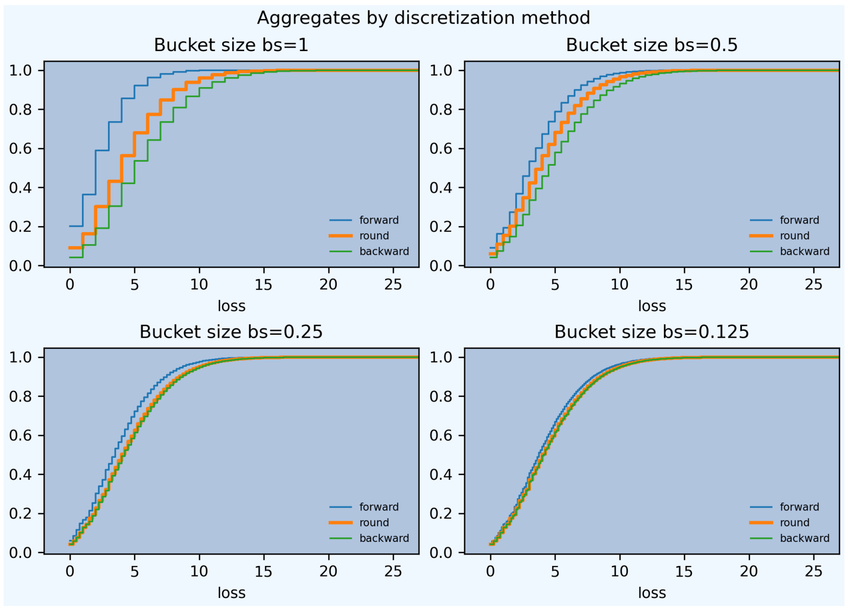
(continued from previous page)

```
In [16]: fig.suptitle('Severity by discretization method for different bucket sizes
↪');
```



Next, create aggregate distributions with a Poisson frequency, mean 4 claims, shown for the same values of `bs`.

```
In [17]: a02 = build(f'agg Num:02 4 claims dsev {dsev} poisson', update=False)

In [18]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.45 + 0.1),
   ....:     constrained_layout=True)
   ....:

In [19]: for bs, ax in zip([1, 1/2, 1/4, 1/8], axs.flat):
   ....:     for k in ['forward', 'round', 'backward']:
   ....:         a02.update(log2=10, bs=bs, sev_calc=k)
   ....:         a02.density_df.p_total.cumsum().\
   ....:             plot(xlim=[-2, 27], lw=2 if  k=='round' else 1,
   ....:            drawstyle='steps-post', label=k, ax=ax)
   ....:         ax.legend(loc='lower right')
   ....:         ax.set(title=f'Bucket size bs={bs}')
   ....:

In [20]: fig.suptitle('Aggregates by discretization method');
```



Note: Setting `drawstyle='steps-post'` joins dots with a step function that jumps on the right (post=afterwards), making the result continuous from the right, appropriate for a distribution. Quantile functions are continuous from the left and should be rendered using `drawstyle='steps-pre'` (before), which puts the jump on the left.

Exact Calculation

The differences $p_k = F((k+1/2)b) - F((k-1/2)b)$ can be computed in three different ways, controlled by the `discretization_calc` option. The options are:

1. `discretization_calc='distribution'` takes differences of the sequence $F((k+1/2)b)$. This results in a potential loss of accuracy in the right tail where the distribution function increases to 1. The resulting probabilities can be no smaller than the smallest difference between 1 and a float. `numpy` reports this as `numpy.finfo(float).epsneg`; it is of the order `1e-16`.
2. `discretization_calc='survival'` takes the negative difference of the sequence $S(k+1/2)b)$ of survival function values. This results in a potential loss of accuracy in the left tail where the survival function increases to 1. However, it provides better resolution in the right.
3. `discretization_calc='both'` attempts to make the best of both worlds, computing:

   ```
   np.maximum(np.diff(fz.cdf(adj_xs)), -np.diff(fz.sf(adj_xs)))
   ```

   This does double the work and is marginally slower.

The update default is `survival`. The calculation method does not generally impact the aggregate distribution when FFTs are used because they compute to accuracy about `1e-16` (there is a 1 in each row and column of F, see num fft). However, the option can be helpful to create a pleasing graph of severity log density.

Truncation and Normalization

The discrete probabilities p_k must be truncated into a finite-length vector to use in calculations. The number of buckets used is set by the `log2` variable, which inputs its base 2 logarithm. The default is `log2=16` corresponding to 65,536 buckets. There are two truncation options, controlled by the `normalize` variable.

1. `normalize=False` simply truncates, possibly resulting in a vector of probabilities that sums to less than 1.
2. `normalize=True` truncates and then normalizes, dividing the truncated vector by its sum, resulting in a vector of probabilities that does sums to 1 (approximately, see floats).

The default is `normalize=True`.

It is obviously desirable for the discrete probabilities to sum to 1. A third option, to put a mass at the maximum loss does not produce intuitive results—since the underlying distributions generally do not have a mass the graphs look wrong.

In general, it is best to use `normalize=True` in cases where the truncation error is immaterial, for example with a thin tailed severity. It is numerically cleaner and avoids issues with quantiles close to 1. When there will be an unavoidable truncation error, it is best to use `normalize=False`. The user needs to be aware that the extreme right tail is understated. The bucket size and number of buckets should be selected so that the tail is accurate where it is being relied upon. See num error analysis for more.

Warning: Avoid using `normalize=True` for thick tail severities. It results in unreliable and hard to interpret estimated mean severity.

Truncation Example

Schaller and Temnov [2008] consider a Poisson-generalized Pareto model for operational risk. They assume an expected claim count equal to 18 and a generalized Pareto with shape 1, scale 12000 and location 7000. This distribution does not have a mean. They want to model the 90th percentile point. They compare using exponential tilting [Grübel and Hermesmeier, 1999] with padding, using up to 1 million `log2=20` buckets. They use a right-hand endpoint of 1 million on the severity. This example illustrates the impact of normalization and shows that padding and tilting have a similar effect.

Setup the base distribution without recomputing. Note infinite severity.

```
In [21]: a = build('agg Schaller:Temnov '
   ....:           '18 claims '
   ....:           'sev 12000 * genpareto 1 + 7000 '
   ....:           'poisson'
   ....:           , update=False)
   ....:

In [22]: qd(a)

      E[X]  CV(X) Skew(X)
X
Freq    18 0.2357  0.2357
Sev    inf
Agg    inf
log2 = 0, bandwidth = na, validation: n/a, not updated.
```

Execute a variety of updates and assemble answer. Compare Schaller and Temnov [2008], Example 4.3.2, p. 197. They estimate the 90th percentile as 3,132,643. In this case, normalizing severity has a material impact; it acts to decrease the tail thickness and hence estimated percentiles.

```
In [23]: import time

In [24]: import pandas as pd
```

(continues on next page)

(continued from previous page)

```
In [25]: updates = {
   ....:     'a': dict(log2=17, bs=100, normalize=True, padding=0 , tilt_
↪vector=None),
   ....:     'b': dict(log2=17, bs=100, normalize=False, padding=0, tilt_
↪vector=None),
   ....:     'c': dict(log2=17, bs=100, normalize=False, padding=1, tilt_
↪vector=None),
   ....:     'd': dict(log2=17, bs=100, normalize=False, padding=2, tilt_
↪vector=None),
   ....:     'e': dict(log2=20, bs=25, normalize=True, padding=1 , tilt_
↪vector=None),
   ....:     'f': dict(log2=20, bs=25, normalize=False, padding=1 , tilt_
↪vector=None),
   ....:     'g': dict(log2=17, bs=100, normalize=False, padding=0, tilt_vector=20␣
↪/ (1<<17))
   ....:     }
   ....:

In [26]: ans = {}

In [27]: for k, v in updates.items():
   ....:     start_time_ns = time.time_ns()
   ....:     a.update(**v)
   ....:     end_time_ns = time.time_ns()
   ....:     ans[k] = [a.q(0.9), a.q(0.95), a.q(0.99),  (-start_time_ns + end_time_
↪ns) / 1e6]
   ....:

In [28]: df = pd.DataFrame(ans.values(), index=ans.keys(), columns=[.9, .95, .99,
↪'millisec'])

In [29]: for k, v in updates['a'].items():
   ....:     df[k] = [v[k] for v in updates.values()]
   ....:

In [30]: df = df.replace(np.nan, 'None')

In [31]: df = df.set_index(['log2', 'bs', 'normalize', 'padding', 'tilt_vector'])

In [32]: df.columns.name = 'percentile'

In [33]: qd(df, float_format=lambda x: f'{x:12,.0f}', sparsify=False, col_space=4)

percentile                                    900.000m     950.000m     990.000m    ␣
↪millisec
log2 bs   normalize padding tilt_vector                                             ␣
↪
17   100  True      0       None             2,789,700    4,280,300    8,936,500    ␣
↪     20
17   100  False     0       None             3,090,100    5,294,500   13,107,100    ␣
↪     20
17   100  False     1       None             3,132,600    5,466,900   13,107,100    ␣
↪     25
17   100  False     2       None             3,132,700    5,467,100   13,107,100    ␣
↪     38
20   25   True      1       None             2,966,775    4,851,225   13,306,850    ␣
↪    255
20   25   False     1       None             3,132,675    5,467,125   23,117,100    ␣
↪    200
17   100  False     0       152.588u         3,132,700    5,467,100   13,107,100    ␣
```

(continues on next page)

(continued from previous page)

```
↪       25
```

Estimating the Bucket Size

Warning: Estimating the bucket size correctly is critical to obtaining accurate results from the FFT algorithm. This section is very important!

The bucket size is estimated as the p-percentile of a moment matched fit to the aggregate. By default $p = 0.999$, but the user can selection another value using the `recommend_p` argument to `update`.

On creation, `Aggregate` automatically computes the theoretic mean, CV, and skewness γ of the requested distribution. Using those values and p the bucket size is estimated as follows.

1. If the CV is infinite the user must input b. An `ValueError` is thrown if no value is provided. Without a standard deviation there is no way to gauge the scale of the distribution. Note that the CV is automatically infinite if the mean does not exist.
2. Else if the CV is finite and $\gamma < 0$, fit a normal approximation (matching two moments). Most insurance applications have positive skewness.
3. Else if the CV is finite and $0 < \gamma < \infty$, fit shifted lognormal and gamma distributions (matching three moments), and a normal distribution.
4. Else if the CV is finite but skewness is infinite, fit lognormal, gamma, and normal distributions (two moments).
5. Compute b' as the greatest of any fit distribution p-percentile (usually the lognormal).
6. If all severity components are limited, compute the maximum limit, m, otherwise set $m = 0$.
7. Take $b = \max(b', m)/n$, where n is the number of buckets.
8. If $b \ge 1$ round up to 1, 2, 5, 10, 20, 100, 200, etc., and return. Else if $b < 1$ return the smallest power of 2 greater than b (e.g., 0.2 rounds to 0.25, 0.1 to 0.125).

Step 8 ensures that $b \ge 1$ is a reasonable looking round number and is an exact float when $b \le 1$. The algorithm performs well in practice, though it can under-estimate b for thick-tailed severities. The user should always look at the diagnostics *Aggregate Quick Diagnostics*.

Occurrence Reinsurance and Loss Picks

If specific layer loss picks are selected, the adjustment occurs immediately after the gross severity is computed in Step 2.

Occurrence reinsurance is applied after loss pick adjustment and before step 3.

Aggregate reinsurance is applied at the very end of the algorithm.

5.4.4 Fourier Transform Convolution Algorithm

> We come now to reality. The truth is that the digital computer has totally defeated the analog computer. The input is a sequence of numbers and not a continuous function. The output is another sequence of numbers, whether it comes from a digital filter or a finite element stress analysis or an image processor. **The question is whether the special ideas of Fourier analysis still have a part to play, and the answer is absolutely yes.** [Strang, 1986]

The previous section quoted Strang in support of discrete models. Here we complete his quote in support of using Fourier analysis, born in application to continuous functions, in a discrete setting.

The `aggregate` Convolution Algorithm: Details

This section expands on each step in the **FFT algorithm** presented at the end of *The aggregate Convolution Algorithm* in more detail. Recall, the four steps:

1. Discretize the severity cdf.
2. Apply the FFT to discrete severity.
3. Apply the frequency pgf to the FFT.
4. Apply the inverse FFT to create a discretized approximation to the aggregate distribution.

Algorithm Objective

Compute a discrete approximation to the aggregate distribution

$$A = X_1 + \cdots + X_N,$$

under the assumption that X_i are iid like X and N is independent of X_i.

Algorithm Inputs

1. Severity distribution (cdf, sf, and moments), optionally including loss pick and occurrence reinsurance adjustments.
2. Frequency distribution (probability generating function $\mathscr{P}(z) := \mathsf{E}[z^N]$, and moments).
3. Number of buckets, expressed as log base 2, $n = 2^{\mathit{log2}}$.
4. Bucket size, b.
5. Severity calculation method: `round`, `forwards`, or `backwards`
6. Discretization calculation method: `survial`, `distribution`, or `both`.
7. Normalization parameter, `True` or `False`.
8. Padding parameter, an integer $d \ge 0$.
9. Tilt parameter, a real number $\theta \ge 0$.
10. Remove "fuzz" parameter, `True` or `False`

Usually either tilting or padding is applied to manage aliasing, but not both. When both are requested, tilting is applied first and the result is zero padded.

Default and Reasonable Parameter Values

The severity and frequency distributions are required. Defaults and reasonable ranges for the other parameters are as follows. Set $x_{max} = bn$ to be the range of the discretized output.

3. *log2* $= 16$, with a reasonable range $3 \le$ *log2* $\le 28 - d$ (on a 64-bit computer with 32GB RAM).
4. Estimating the bucket size is quite involved and is described in num rec bucket.
5. Severity calculation `round`, see num rounding default.
6. Discretization `survival`, see num exact calculation.
7. Normalization defaults to *True*. It should only be used if it is immaterial, in the sense that $1 - F(x_{max})$ is small. See num normalization.

8. Padding equals 1, meaning severity is doubled in size and padded with zeros. Padding equals to 2, which quadruples the size of the severity vector, is sometimes necessary. Schaller and Temnov [2008] report requiring padding equal to 2 in empirical tests with very high frequency and thick tailed severity.
9. Tilt equals 0 (no tilting applied). Embrechts and Frei [2009] recommend $\theta n \le 20$. Schaller and Temnov [2008] section 4.2 discuss how to select θ to the decrease in aliasing error and impact on numerical precision. Padding is an effective way to manage aliasing, but no more so than padding most circumstances. We prefer the simpler padding approach.
10. Remove fuzz is `True`.

Algorithm Steps

The default steps are shown next, followed by further explanation.

1. If frequency is identically zero, then $(1, 0, \dots)$ is returned with no further calculation.
2. If frequency is identically one, then the discretized severity is returned with no further calculation.
3. If needed, estimate the bucket size, num rec bucket.
4. Discretize severity into a vector $\mathsf{p} = (p_0, p_1, \dots, p_{n-1})$, see num discretization. This step may include normalization.
5. Tilt severity, $p_k \leftarrow p_k e^{-k\theta}$.
6. Zero pad the vector p to length 2^{log2+d} by appending zeros, to produce x.
7. Compute $\mathsf{z} := \mathsf{FFT}(\mathsf{x})$.
8. Compute $\mathsf{f} := \mathscr{P}(\mathsf{z})$.
9. Compute the inverse FFT, $\mathsf{y} := \mathsf{IFFT}(\mathsf{f})$.
10. Take the first n entries in y to obtain $\mathsf{a} := \mathsf{y}[0:n]$.
11. Aggregate reinsurance is applied a if applicable.

Theory: Why the Algorithm Works

This section explains why the output output $\mathsf{a} = (a_0, \dots, a_{m-1})$ has a_k very close to $\Pr(A = kb)$.

Fourier transforms provide an alternative way to represent a distribution function. The [Wikipedia](https://en.wikipedia.org/wiki/Fourier_transform) article says:

> The Fourier transform of a function is a complex-valued function representing the complex sinusoids that comprise the original function. For each frequency, the magnitude (absolute value) of the complex value represents the amplitude of a constituent complex sinusoid with that frequency, and the argument of the complex value represents that complex sinusoid's phase offset. If a frequency is not present, the transform has a value of 0 for that frequency. The Fourier inversion theorem provides a synthesis process that recreates the original function from its frequency domain representation.
>
> Functions that are localized in the time domain have Fourier transforms that are spread out across the frequency domain and vice versa, a phenomenon known as the uncertainty principle. The critical case for this principle is the Gaussian function: the Fourier transform of a Gaussian function is another Gaussian function.
>
> Generalizations include the discrete-time Fourier transform (DTFT, group Z), the discrete Fourier transform (DFT, group $Z \pmod N$) and the Fourier series or circular Fourier transform (group = S^1, the unit circle being a closed finite interval with endpoints identified). The latter is routinely employed to handle periodic functions. The fast Fourier transform (FFT) is an algorithm for computing the DFT.

The Fourier transform (FT) of a distribution function F is usually written $\hat{F}$. The FT contains the same information as the distribution and there is a dictionary back and forth between the two, using the inverse FT. Some computations with distributions are easier to perform using their FT, which is what makes them useful. The FT is like exponentiation for distributions. The exponential and log functions turn (difficult) multiplication into (easy) addition

$$e^a \times e^b = e^{a+b}.$$

FTs turn difficult convolution of distributions (addition of the corresponding random variables) into easy multiplication of Fourier transforms. If X_i are random variables, $X = X_1 + X_2$, and F_X is the distribution of X, then

$$\widehat{F_{X_1+X_2}}(t) = \widehat{F_{X_1}}(t) \times \widehat{F_{X_2}}(t),$$

where the righthand side is a product of functions. Computing the distribution of a sum of random variables is complicated because you have to consider all different ways an outcome can be split, but it is easy using FTs. Of course, this depends on it being easy to compute the FT and its inverse—which is where FFTs come in.

There are three things going on here:

1. **Fourier transform** of a function,
2. **Discrete** Fourier transform of an infinite sequence, and
3. **Fast** Fourier transform of a finite vector.

Discrete Fourier transforms are a discrete approximation to continuous FTs, formed by sampling at evenly spaced points. The DFT is a sequence, rather than a function. It retains the convolution property of FTs. They are sometimes called discrete cosine transforms (DCT).

The **Fast** Fourier transform refers to a very fast way to compute *finite* discrete FTs, which are applied to finite samples of FTs. General usage blurs the distinction between discrete FTs and their computation, and uses FFT as a catchall for both.

Thus, there are four-steps from the continuous to the finite discrete computational strategy (notation explained below):

1. Analytic domain:

$$f \to \hat{f} \to \mathscr{P} \circ \hat{f} \to \widehat{\mathscr{P} \circ \hat{f}} =: g$$

2. Discrete approximation:

$$f \to f_b \to \hat{f}_b \to \mathscr{P} \circ \hat{f}_b \to \widehat{\mathscr{P} \circ \hat{f}_b} =: g_b$$

3. Finite discrete approximation:

$$f \to f_{b,n} \to \hat{f_{b,n}} \to \mathscr{P} \circ \hat{f_{b,n}} \to \widehat{\mathscr{P} \circ \hat{f_{b,n}}} =: g_{b,n}$$

4. Finite discrete approximation, periodic inversion:

$$f \to f_{b,n} \to \hat{f_{b,n}} \to \mathscr{P}_m \circ \hat{f_{b,n}} \to \widehat{\mathscr{P}_m \circ \hat{f_{b,n}}} =: g_{b,n,m}$$

Here is the rationale for each step.

- Step 1 to 2: **discretize** f because we are working in a digital world, not an analog/analyic one (Strang quote) and because the answers are often not continuous distributions. Discretize at multiples of a sampling interval b. The sampled distribution (which no longer has a density) is The sampling rate is $1/b$ samples per unit. The sampled distribution (which no longer has a density) is

$$f_b = \sum_k p_k \delta_{kb}.$$

  f_b has Fourier transform

$$\hat{f}_b(t) = \sum_k p_k e^{-2\pi i k b t}.$$

  If $\hat{f}$ is know analytically it can be sampled directly, see the stable example below. However, many relevant f do not have analytic FTs, e.g., lognormal. At this point, f_b is still defined of $\mathbb{R}$.

- Step 2 to 3: **truncate** and take a **finite** discretization because we are working on a digital computer with finite memory.

  $$f_{b,n} = \sum_{k=0}^{n-1} p_k \delta_{kb}.$$

  Let $P = nb$. Now $f_{b,n}$ is non-zero only on $[0, P)$. Finite discretization combined with an assumption of P periodicity enables the use of **FFTs** to compute $f_{b,n} \to \hat{f_{b,n}}$. (In order for a Fourier series to be P-periodic, it can only weight frequencies that are a multiple of $1/P$ since $\exp(2\pi i(x + kP)t) = \exp(2\pi ixt)$ for all integers k iff $\exp(2\pi ikPt) = 1$ iff Pt is an integer. Take the integer to be 1; higher values correspond to aliasing. Hence Shannon-Nyquist and bandwidth limited functions etc.) Sampling $\widehat{f_{b,n}}(t)$ at $t = 0, 1/P, \dots, (n-1)/P$, requires calculating

  $$\hat{f_{b,n}}(\tfrac{l}{P}) = \sum_k p_k e^{-2\pi ikb\frac{l}{P}} = \sum_k p_k e^{-\frac{2\pi i}{n}kl}$$

  which is exactly what FFTs compute very quickly.

- Step 3 to 4: **finite convolution**, $\mathscr{P}_m$ is computed with a sample of length $m \ge n$, i.e., padding, to control aliasing. We can also use exponential tilting (which must be done in the f domain). $\mathscr{P}_m \circ \hat{f_{b,n}}$ is the application of a function to a vector, element-by-element and is easy to compute. $\mathscr{P}_m \circ \hat{f_{b,n}} \to \widehat{\mathscr{P}_m \circ \hat{f_{b,n}}}$ can be computed using FFTs, whereas inverting $\mathscr{P} \circ \hat{f_{b,n}}$ would usually be very difficult because it usually has infinite support. The price for using FFTs is assuming g is P-periodic, i.e., introducing aliasing error. For simplicity, assume $m = n$ by padding the samples in Step 2.

  Now we can use the inverse DFT to recover g at the values kb:

  $$g(kb) = \sum_l \hat{g}(\tfrac{l}{P}) e^{2\pi ikb\frac{l}{P}} \tag{5.1}$$

  $$= \sum_l \hat{g}f(\tfrac{l}{P}) e^{\frac{2\pi i}{n}kl} \tag{5.2}$$

  However, this is an infinite sum (step 3), and we are working with computers, so it needs to be truncated (step 4). What is

  $$\sum_{l=0}^{n-1} \hat{g}(\tfrac{l}{P}) e^{\frac{2\pi i}{n}kl}?$$

  It is an inverse DFT, that FFTs compute with alacrity. What does it equal?

  Define $g_P(x) = \sum_k g(x + kP)$ to be the P-periodic version of g. If g has finite support contained in $[0, P)$ then $g_P = g$. If that is not the case there will be wrapping spill-over, see PICTURE.

  Now

  $$\hat{g}(\tfrac{l}{P}) := \int_{\mathbb{R}} g(x) e^{-2\pi ix\frac{l}{P}} dx \tag{5.3}$$

  $$= \sum_k \int_{kP}^{(k+1)P} g(x) e^{-2\pi ix\frac{l}{P}} dx \tag{5.4}$$

  $$= \sum_k \int_0^P g(x + kP) e^{-2\pi i(x+kP)\frac{l}{P}} dx \tag{5.5}$$

  $$= \int_0^P \sum_k g(x + kP) e^{-2\pi ix\frac{l}{P}} dx \tag{5.6}$$

  $$= \int_0^P g_P(x) e^{-2\pi ix\frac{l}{P}} dx \tag{5.7}$$

  $$= \hat{g_P}(\tfrac{l}{P}) \tag{5.8}$$

and therefore, arguing backwards and assuming that $\hat{g}$ is quickly decreasing, for large enough n,

$$\sum_{l=0}^{n-1} \hat{g}(\tfrac{l}{P}) e^{\frac{2\pi i}{n}kl} \approx \sum_{l} \hat{g}(\tfrac{l}{P}) e^{\frac{2\pi i}{n}kl} \tag{5.9}$$

$$= \sum_{l} \hat{g_P}(\tfrac{l}{P}) e^{\frac{2\pi i}{n}kl} \tag{5.10}$$

$$= g_P(kb) \tag{5.11}$$

Thus the partial sum we can easily compute on the left approximates g_P and in favorable circumstances it is close to g.

There are four sources of error in the FFT algorithm. They can be controlled by different parameters:

1. Discretization error $f \leftrightarrow f_b$ (really $\hat{f} \leftrightarrow \hat{f}_b$): replacing the original distribution with a discretized approximation, controlled by decreasing the bucket size.
2. Truncation error $\hat{f}_b \leftrightarrow \widehat{f_{b,n}}$: shrinking the support of the severity distribution by right truncation, controlled by increasing the bucket size and/or increasing the number of buckets.
3. Aliasing error $\widehat{\mathscr{P} \circ \widehat{f_{b,n}}} \leftrightarrow \widehat{\mathscr{P}_m \circ \widehat{f_{b,n}}}$: expect g_k get $\sum_l g_{k+ln}$: working with only finitely many frequencies in the Fourier domain which results in visible the aggregate wrapping, controlled by padding or tilting severity.
4. FFT algorithm: floating point issues, underflow and (rarely) overflow, hidden by removing numerical "fuzz" after the algorithm has run.

To summarize:

- If we know $\hat{f}$ analyically, we can use this method to estimate a discrete approximation to f. We are estimating $f_P(kb)$ not $f(kb)$, so there is always aliasing error, unless f actually has finite support.
- If f is actually discrete, the only error comes from truncating the Fourier series. We can make this as small as we please by taking enough terms in the series. This case is illustrated for the Poisson distribution. This method is also applied by `aggregate`: the "analytic" chf is $\mathscr{P}_N(M_X(t))$, where $M_X(t)$ is the sum of exponentials given above.
- When $\hat{f}$ is known we have a choice between discretizing in the space (loss) or time domain.
- If f is not discrete, there is a discretization and potentially aliasing error. We can control the former with high frequency (small b) sampling. We control the latter with large $P = nb$, arguing for large n or large b (in conflict to managing discretization error).

Using FFT to Invert Characteristic Functions

The use of FFTs to recover the aggregate at the end of Step 4 is entirely generic. It can be used to invert any characteristic function. In this section we provide some of examples.

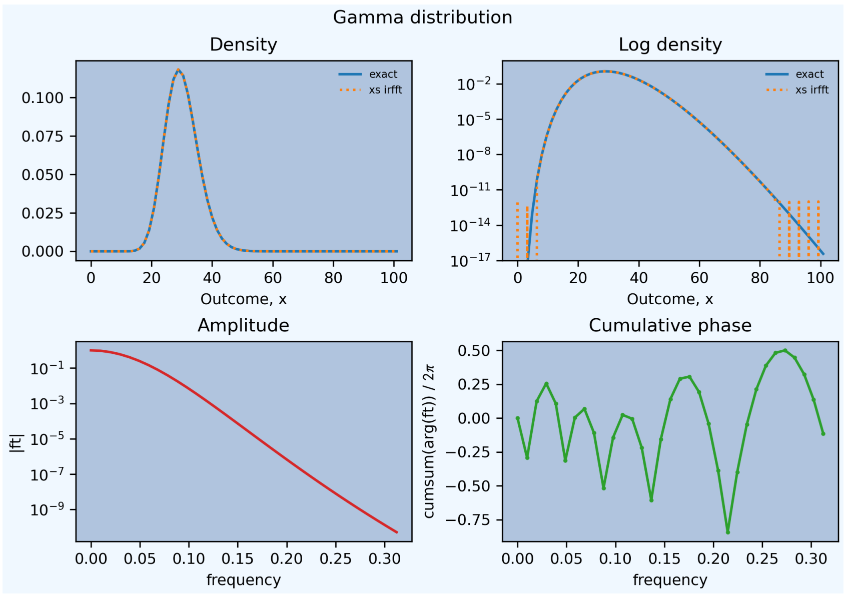
Invert a gamma distribution from a sample of its characteristic function and compare with the true density. These plots show the inversion is extremely accurate over a very wide range. The top right plot compares the log density, highlighting differences only in the extreme tails.

```
In [1]: from aggregate.extensions import ft_invert

In [2]: import scipy.stats as ss

In [3]: import matplotlib.pyplot  as plt

In [4]: df = ft_invert(
   ...:           log2=6,
   ...:           chf=lambda alpha, t: (1 - 1j * t) ** -alpha,
   ...:           frz_generator=ss.gamma,
   ...:           params=[30],
   ...:           loc=0,
```

(continues on next page)

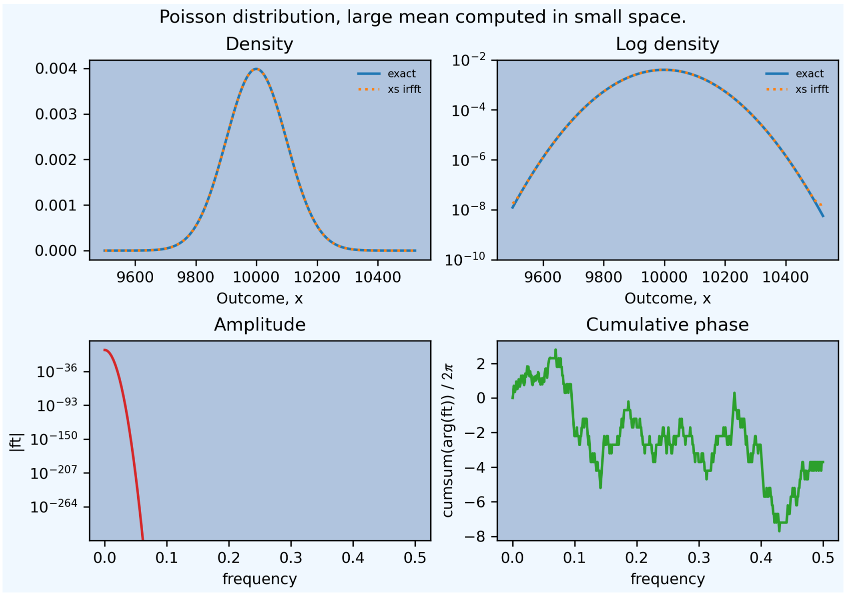
(continued from previous page)

```
   ...:            scale=1,
   ...:            xmax=0,
   ...:            xshift=0,
   ...:            suptitle='Gamma distribution')
   ...:
```



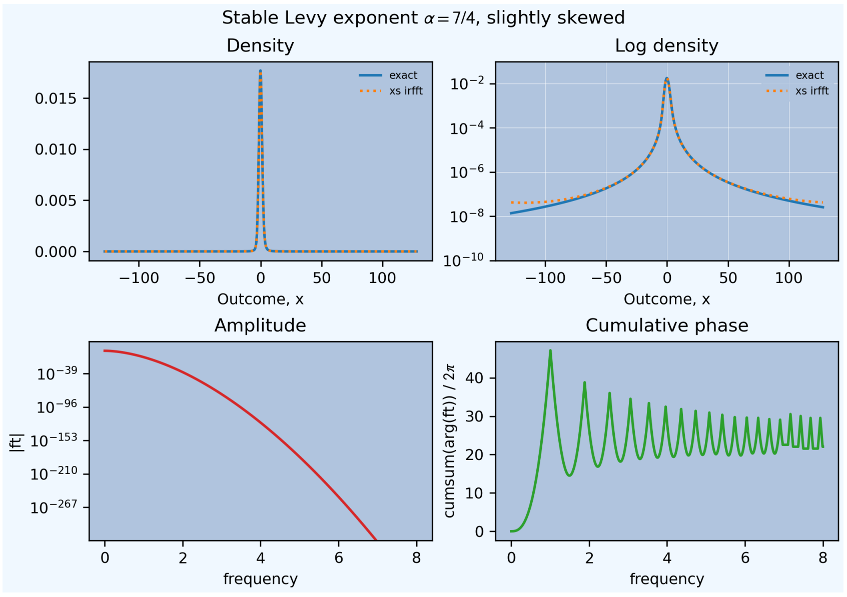
Invert a Poisson distribution with a very high mean. This is an interesting case, because we do not need space for the whole answer, just the effective range of the answer. We can use periodicity to "move" the answer to the right x range. This example reproduces a Poisson with mean 10,000. The standard deviation is only 100 and so the effective rate of the distribution (using the normal approximation) will be about 9500 to 10500. Thus a satisfactory approximation can be obtained with only $2^{10} = 1024$ buckets.

```
In [5]: import aggregate.extensions.ft as ft

In [6]: df = ft.ft_invert(
   ...:            log2=10,
   ...:            chf=lambda en, t: np.exp(en * (np.exp(1j * t) - 1)),
   ...:            frz_generator=ss.poisson,
   ...:            params=[10000],
   ...:            loc=0,
   ...:            scale=None, # for freq dists, scaling does not apply
   ...:            xmax=None,  # for freq dists want bs = 1, so xmax=1<<log2
   ...:            xshift=9500,
   ...:            suptitle='Poisson distribution, large mean computed in small␣
↪space.')
   ...:
```



Invert a stable distribution. Here there is more aliasing error because the distribution is so thick tailed. There is also more on the left than right because of the asymmetric beta parameter.

```
In [7]: def levy_chf(alpha, beta, t):
   ...:     Φ = np.tan(np.pi * alpha / 2) if alpha != 1 else -2 / np.pi * np.
↪log(np.abs(t))
   ...:     return np.exp(- np.abs(t) ** alpha * (1 - 1j * beta * np.sign(t) * Φ))
   ...:

In [8]: df = ft.ft_invert(
   ...:         log2=12,
   ...:         chf=levy_chf,
   ...:         frz_generator=ss.levy_stable,
   ...:         params=[1.75, 0.3],  # alpha, beta
   ...:         loc=0,
   ...:         scale=1.,
   ...:         xmax=1<<8,
   ...:         xshift=-(1<<7),
   ...:         suptitle='Stable Levy exponent $\\alpha=7/4$, '
   ...:         'slightly skewed')
   ...:

In [9]: f = plt.gcf()

In [10]: ax = f.axes[1]

In [11]: ax.grid(lw=.25, c='w');
```



Fast Fourier Transforms

The trick with FFTs is *how* they are computed. *What* they compute is very straightforward and given by a simple matrix multiplication.

The FFT of the $m \times 1$ vector $\mathsf{x} = (x_0, \dots, x_{m-1})$ is just another $m \times 1$ vector $\hat{\mathsf{x}}$ whose jth component is

$$x_j = \sum_{k=0}^{m-1} x_k \exp(-2\pi i jk/m),$$

where $i = \sqrt{-1}$. The coefficients of $\hat{\mathsf{x}}$ are complex numbers. It is easy to see that $\hat{\mathsf{x}} = \mathsf{F}\mathsf{x}$ where

$$\mathsf{F} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w & \dots & w^{m-1} \\ 1 & w^2 & \dots & w^{2(m-1)} \\ \vdots & & & \vdots \\ 1 & w^{m-1} & \dots & w^{(m-1)^2} \end{pmatrix}$$

is a matrix of complex roots of unity and $w = \exp(-2\pi i/m)$. This shows there is nothing inherently mysterious about an FFT. The trick is that there is a very efficient algorithm for computing the matrix multiplication [Press *et al.*, 1992]. Rather than taking time proportional to m^2, as one would expect, it can be computed in time proportional to $m\log(m)$. For large values of m, the difference between $m\log(m)$ and m^2 time is the difference between practically possible and practically impossible.

The inverse FFT to recovers x from its transform $\hat{\mathsf{x}}$. The inverse FFT is computed using the same equation as the FFT with F^{-1} (matrix inverse) in place of F. It is easy to see that inverse equals

$$\mathsf{F}^{-1} = \frac{1}{m}\begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w^{-1} & \dots & w^{-(m-1)} \\ 1 & w^2 & \dots & w^{2(m-1)} \\ \vdots & & & \vdots \\ 1 & w^{-(m-1)} & \dots & w^{-(m-1)^2} \end{pmatrix}.$$

The (j, j) element of $m\mathsf{F}\mathsf{F}^{-1}$ is

$$\sum_g w^{jg} w^{-jg} = \sum_g 1 = m$$

and the $(j,k),\ j\neq k$ element is

$$\sum_g w^{jg}w^{-gk} = \sum_g w^{g(j-k)} = 0.$$

The inversion process can also be computed in $m\log(m)$ time because the matrix equation is the same.

How does the FFT compute convolutions? Given two probability vectors for outcomes $k=0,1,\dots,n-1$, say $\mathsf{p}=(p_0,\dots,p_{n-1})$ and $\mathsf{q}=(q_0,\dots,q_{n-1})$, the product of the k th elements of the FFTs equals

$$\left(\sum_g p_g w^{gk}\right)\left(\sum_h p_h w^{hk}\right) = \sum_{m=0}^{n-1}\left(\sum_{\substack{g,h\\ g+h\equiv m(n)}} p_g q_h\right) w^{km}$$

is the k th element of the FFT of the wrapped convolution of p and q. For example, if $n=4$ and $m=0$, the inner sum on the right equals

$$p_0q_0 + p_1q_3 + p_2q_2 + p_3q_1$$

which can be interpreted as

$$p_0q_0 + p_1q_{-1} + p_2q_{-2} + p_3q_{-3}$$

in arithmetic module n.

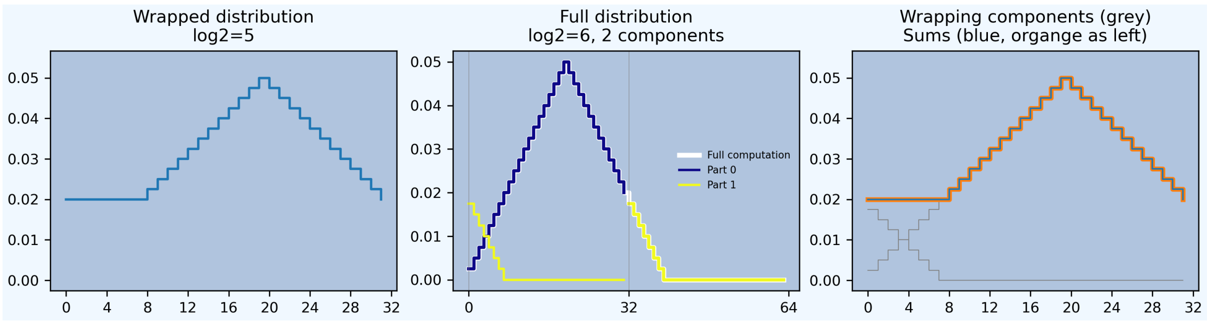
In the convolution algorithm, the product of functions $\widehat{F_{X_1}}\times\widehat{F_{X_2}}$ is replaced by the component-by-component product of two vectors, which is easy to compute. Thus, to convolve two discrete distributions, represented as $\mathsf{p}=(p_0,\dots,p_{m-1})$ and $\mathsf{q}=(q_0,\dots,q_{m-1})$ simply

- Take the FFT of each vector, $\hat{\mathsf{p}}=\mathsf{F}\mathsf{p}$ and $\hat{\mathsf{q}}=\mathsf{F}\mathsf{q}$
- Compute the component-by-component product $\mathsf{z}=\hat{\mathsf{p}}\hat{\mathsf{q}}$
- Compute the inverse FFT $\mathsf{F}^{-1}\mathsf{z}$.

The answer is the exact convolution of the two input distributions, except that sum values wrap around: the extreme right tail re-appears as probabilities around 0. This problem is called aliasing (the same as the wagon-wheel effect in old Westerns), but it can be addressed by padding the input vectors.

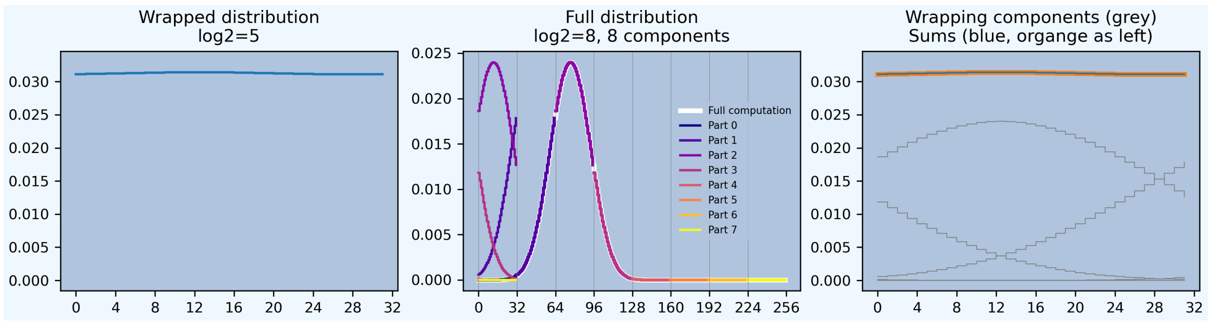
Here is a simple example of wrapping, using a compound Poisson distribution with an expected claim count of 10 and severity taking the values 0, 1, 2, 3, or 4 equally often. The aggregate has a mean of 20 and is computed using only 32 buckets. This is not enough space, and the right hand part of the distribution wraps around. The components are shown in the middle and how they combine on the right.

```
In [12]: ft.fft_wrapping_illustration(ez=10, en=2)
```



The next figure illustrates more extreme FFT wrapping. It shows an attempt to model a compound Poisson distribution with a mean of 80 using only 32 buckets. The result is the straight line on the left. The middle plot shows the true distribution and the vertical slices of width 32 that are combined to get the total. These are shown shifted on the left. The right plot zooms into the rate `0:32`, and shows how the wrapped components sum to the result on the left. This is a good example of how FFT methods can fail and can appear to give inexplicable results.

```
In [13]: ft.fft_wrapping_illustration(ez=10, en=8)
```



It is not necessary to understand the details of FTs to use `aggregate` although they are fascinating, see for example Körner [2022]. In probability, the moment generating functions and characteristic function are based on FTs. They are discussed in any serious probability text.

FFT Routines

Computer systems offer a range of FFT routines. `aggregate` uses two functions from `scipy.fft` called `scipy.fft.rfft()` and `scipy.fft.irfft()`. There are similar functions in `numpy.fft`. They are tailored to taking FFTs of vectors of real numbers (as opposed to complex numbers). The FFT routine automatically handles padding the input vector. The inverse transform returns real numbers only, so there is no need to take the real part to remove noise-level imaginary parts. It is astonishing that the whole `aggregate` library pivots on a single line of code:

```
agg_density = rifft(\mathscr P(rfft(p)))
```

Obviously, a lot of work is done to marshal the input, but this line is where the magic occurs.

The FFT routines are accurate up to machine noise, of order `1e-16`. The noise can be positive or negative—the latter highly undesirable in probabilities. It appears random and does not accumulate undesirably in practical applications. It is best to strip out the noise, setting to zero all values with absolute value less than machine epsilon (`numpy.finfo(float).esp`). The `remove_fuzz` option controls this behavior. It is set `True` by default. CHECK SURE?

5.4.5 Floating Point Arithmetic and Rounding Errors

The internal workings of computer floating point arithmetic can cause unexpected problems. You can read no further in this section if you promise to obey

Warning: Only use a bucket size b with an exact floating point representation. It must have an exact binary representation as a fraction a/b where b is a power of two.

1/3, 1/5 and 1/10 are **not** binary floats.

For those who choose to continue, this section presents random selection of results about floats that tripped me up as I wrote `aggregate`.

Floating point arithmetic is not associative!

```
In [1]: x = .1 + (0.6 + 0.3)

In [2]: y = (0.1 + 0.6) + 0.3

In [3]: x, x.as_integer_ratio(), y, y.as_integer_ratio()
Out[3]: (0.9999999999999999, (9007199254740991, 9007199254740992), 1.0, (1, 1))
```

This fact can be used to create sequences with nasty accumulating errors.

Exercise Redux

Recall the exercise to compute quantiles of a *dice roll*. `aggregate` produces the consistent results—if we look carefully and account for the foibles of floating point numbers. The case $p = 0.1$ is easy. But the case $p = 1/6$ appears wrong. There are two ways we can model the throw of a dice: with frequency 1 to 6 and fixed severity 1, or as fixed frequency 1 and severity 1 to 6. They give different answers. The lower quantile is wrong in the first case (it equals 1) and the upper quantile in the second (2).

```
In [4]: from aggregate import build, qd

In [5]: import pandas as pd

In [6]: d = build('agg Dice dfreq [1:6] dsev [1]')

In [7]: print(d.q(0.1, 'lower'), d.q(0.1, 'upper'))
1.0 1.0

In [8]: print(d.q(1/6, 'lower'), d.q(1/6, 'upper'))
1.0 2.0

In [9]: d2 = build('agg Dice2 dfreq [1] dsev [1:6]')

In [10]: print(d2.q(1/6, 'lower'), d2.q(1/6, 'upper'))
1.0 2.0
```

These differences are irritating! The short answer is to adhere to the warning above.

Here's the long answer, if you want to know. Looking at the source code shows that the quantile function is implemented as a previous or next look up on a dataframe of distinct values of the cumulative distribution function. These two dataframes for the different dice models are:

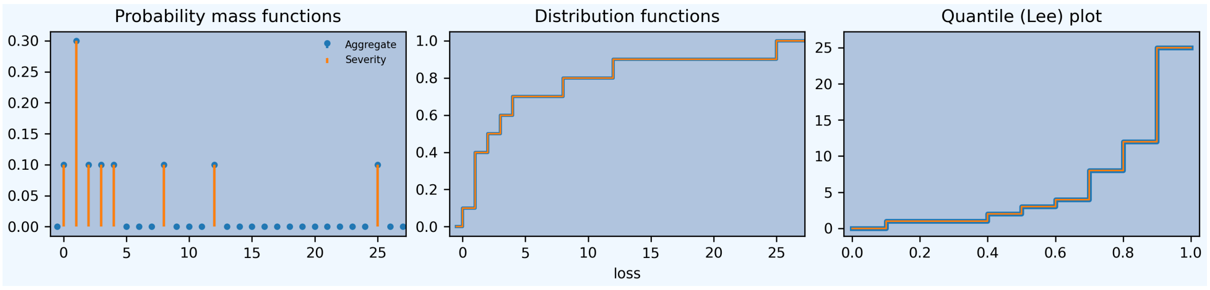
```
In [11]: ff = lambda x: f'{x:.25g}'

In [12]: qd(d.density_df.query('p_total > 0')[['p', 'F']], float_format=ff)

                                p                           F
loss
1.000 0.1666666666666666574148081 0.1666666666666666574148081
2.000 0.1666666666666666296592325  0.3333333333333333259318465
3.000 0.1666666666666666296592325 0.4999999999999999888977698
4.000 0.1666666666666666296592325   0.666666666666666651863693
5.000 0.1666666666666666574148081 0.8333333333333331482961626
6.000 0.1666666666666666574148081 0.9999999999999997779553951

In [13]: qd(d2.density_df.query('p_total > 0')[['p', 'F']], float_format=ff)

                                p                           F
loss
1.000  0.166666666666666740681535  0.166666666666666740681535
2.000 0.1666666666666666296592325 0.3333333333333333703407675
3.000 0.1666666666666666296592325                         0.5
4.000  0.166666666666666740681535  0.666666666666666740681535
5.000  0.166666666666666740681535   0.833333333333333348136307
6.000   0.166666666666666651863693                          1

In [14]: print(f'\n{d.cdf(1):.25f} < {1/6:.25f} < 1/6 < {d2.cdf(1):.25f}')

0.1666666666666666574148081 < 0.1666666666666666574148081 < 1/6 < 0.
↪1666666666666667406815350
```

Based on these numbers, the reported quantiles are correct. $p = 1/6$ is strictly greater than `d.cdf(1)` and strictly less than `d2.cdf(1)`, as shown in the last row! `d` and `d2` are different because the former runs through the FFT routine to convolve the trivial severity, whereas the latter does not.

Exercise

X is a random variable defined on a sample space with ten equally likely events. The event outcomes are $0, 1, 1, 1, 2, 3, 4, 8, 12, 25$. Compute $\mathsf{VaR}_p(X)$ for all p.

```
In [15]: a = build('agg Ex.50 dfreq [1] '
   ....:             'dsev [0 1 2 3 4 8 12 25] [.1 .3 .1 .1 .1 .1 .1 .1]')
   ....:

In [16]: a.plot()

In [17]: print(a.q(0.05), a.q(0.1), a.q(0.2), a.q(0.4),
   ....:    a.q(0.4, 'upper'), a.q(0.41), a.q(0.5))
   ....:
0.0 1.0 1.0 1.0 2.0 2.0 2.0

In [18]: qd(a.density_df.query('p_total > 0')[['p', 'F']],
   ....:      float_format=ff)
   ....:

                                  p                            F
loss
0.000  0.09999999999999997779553951 0.09999999999999997779553951
1.000    0.300000000000000044408921  0.400000000000000022044605
2.000  0.09999999999999997779553951                          0.5
3.000  0.09999999999999997779553951  0.59999999999999997779553395
4.000  0.09999999999999997779553951   0.6999999999999999555910790
8.000   0.0999999999999999639177517  0.79999999999999999333866185
12.000 0.09999999999999997779553951   0.899999999999999991182158
25.000   0.100000000000000088817842                            1
```



Solution. On the graph, fill in the vertical segments of the distribution function. Draw a horizontal line at height p and find its intersection with the completed graph. There is a unique solution for all p except $0.1, 0.4, 0.5, \dots, 0.9$. Consider $p = 0.4$. Any x satisfying $\mathsf{Pr}(X < x) \le 0.4 \le \mathsf{Pr}(X \le x)$ is a 0.4-quantile. By inspection the solutions are $1 \le x \le 2$. VaR is defined as the lower quantile, $x = 1$. The 0.41 quantile is $x = 2$. VaRs are not interpolated in this problem specification. The loss 25 is the p-VaR for any $p > 0.9$. The apparently errant numbers for aggregate (the upper quantile at 0.1 equals 2) are explained by the float representations. The float representation of `0.4` is `3602879701896397/9007199254740992` which actually equals `0.400000000000000022044605`.

5.5 Distortions and Spectral Risk Measures

Objectives: Introduce distortion functions and spectral risk measures.

Audience: Readers looking for a deeper technical understanding.

Prerequisites: Knowledge of probability and calculus; insurance terminology.

See also: 5_x_distortion_calculations.

Contents:

5.5.1 Helpful References

- Mildenhall and Major [2022]
- The text in this section is derived from Major and Mildenhall [2020].
- Mildenhall [2022]

5.5.2 Distortion Function and Spectral Risk Measures

We define SRMs and recall results describing their different representations. By De Waegenaere *et al.* [2003] SRMs are consistent with general equilibrium and so it makes sense to consider them as pricing functionals. The SRM is interpreted as the (ask) price for an insurer-written risk transfer.

Definition. A **distortion function** is an increasing concave function $g : [0, 1] \to [0, 1]$ satisfying $g(0) = 0$ and $g(1) = 1$.

A **spectral risk measure** ρ_g associated with a distortion g acts on a non-negative random variable X as

$$\rho_g(X) = \int_0^\infty g(S(x))dx.$$

The simplest distortion if the identity $g(s) = s$. Then $\rho_g(X) = \mathsf{E}[X]$ from the integration-by-parts identity

$$\int_0^\infty S(x)\,dx = \int_0^\infty xdF(x).$$

Other well-known distortions include the **proportional hazard** $g(s) = s^r$ for $0 < r \le 1$, its **dual** $g(s) = 1-(1-s)^r$ for $r \ge 1$, and the **Wang transform** $g(s) = \Phi(\Phi^{-1}(s) + \lambda)$ for $\lambda \ge 0$, Wang [1995].

Since g is concave $g(s) \ge 0g(0) + sg(1) = s$ for all $0 \le s \le 1$, showing ρ_g adds a non-negative margin.

Going forward, g is a distortion and ρ is its associated distortion risk measure. We interpret ρ as a pricing functional and refer to $\rho(X)$ as the price or premium for insurance on X.

SRMs are **translation invariant**, **monotonic**, **subadditive**, and **positive homogeneous**, and hence **coherent**, Acerbi [2002]. In addition, they are **law invariant** and **comonotonic additive**. In fact, all such functionals are SRMs. As well has having these properties, SRMs are powerful because we have a complete understanding of their representation and structure, which we summarize in the following theorem.

Theorem. Subject to ρ satisfying certain continuity assumptions, the following are equivalent.

1. ρ is a law invariant, coherent, comonotonic additive risk measure.
2. $\rho = \rho_g$ for a concave distortion g.
3. ρ has a representation as a weighted average of TVaRs for a measure μ on $[0,1]$: $\rho(X) = \int_0^1 \mathsf{TVaR}_p(X)\mu(dp)$.
4. $\rho(X) = \max_{\mathsf{Q}\in\mathscr{Q}} \mathsf{E}_{\mathsf{Q}}[X]$ where $\mathscr{Q}$ is the set of (finitely) additive measures with $\mathsf{Q}(A) \le g(\mathsf{P}(A))$ for all measurable A.
5. $\rho(X) = \max_{\mathsf{Z}\in\mathscr{Z}} \mathsf{E}[XZ]$ where $\mathscr{Z}$ is the set of positive functions on Ω satisfying $\int_p^1 q_Z(t)dt \le g(1-p)$, and q_Z is the quantile function of Z.

The Theorem combines results from Föllmer and Schied [2011] (4.79, 4.80, 4.93, 4.94, 4.95), Delbaen [2000], Kusuoka [2001], and Carlier and Dana [2003]. It requires that ρ is continuous from above to rule out the possibility $\rho = \sup$. In certain situations, the sup risk measure applied to an unbounded random variable can only be represented as a sup over a set of test measures and not a max. Note that the roles of from above and below are swapped from Föllmer and Schied [2011] because they use the asset, negative is bad, sign convention whereas we use the actuarial, positive is bad, convention.

The relationship between μ and g is given by Föllmer and Schied [2011] 4.69 and 4.70. The elements of $\mathscr{Z}$ are the Radon-Nikodym derivatives of the measures in $\mathscr{Q}$.

The next four sections introduce the idea of layer densities and prove that SRM premium can be allocated to policy in a natural and unique way.

5.5.3 Layer Densities

Risk is often tranched into layers that are then insured and priced separately. Meyers [1996] describes layering in the context of liability increased limits factors and Culp and O'Donnell [2009], Mango *et al.* [2013] in the context of excess of loss reinsurance.

Define a layer y excess x by its payout function $I_{(x,x+y]}(X) := (X-x)^+ \wedge y$. The expected layer loss is

$$\begin{aligned}
\mathsf{E}[I_{(x,x+y]}(X)] &= \int_x^{x+y} (t-x)dF(t) + yS(x+y) \\
&= \int_x^{x+y} tdF(t) + tS(t)|_x^{x+y} \\
&= \int_x^{x+y} S(t)\,dt.
\end{aligned}$$

Based on this equation, Wang [1996] points out that S can be interpreted as the layer loss (net premium) density. Specifically, S is the layer loss density in the sense that $S(x) = d/dx(\mathsf{E}[I_{(0,x]}(X)])$ is the marginal rate of increase in expected losses in the layer at x. We use density in this sense to define premium, margin and equity densities, in addition to loss density.

Clearly $S(x)$ equals the expected loss to the cover $1_{\{X>x\}}$. By scaling, $S(x)dx$ is the close to the expected loss for $I_{(x,x+dx]}$ when dx is very small; Bodoff [2007] calls these infinitesimal layers.

Wang [1996] goes on to interpret

$$\int_x^{x+y} g(S(t))\,dt$$

as the layer premium and hence $g(S(x))$ as the layer premium density. We write $P(x) := g(S(x))$ for the premium density.

We can decompose X into a sum of thin layers. All these layers are comonotonic with one another and with X, resulting in an additive decomposition of $\rho(X)$, since ρ is comonotonic additive. The decomposition mirrors the definition of ρ as an integral.

The amount of assets a available to pay claims determines the quality of insurance, and premium and expected losses are functions of a. Premiums are well-known to be sensitive to the insurer's asset resources and solvency, Phillips *et al.* [1998]. Assets may be infinite, implying unlimited cover. When a is finite there is usually some chance of default. Using the layer density view, define expected loss $\bar{S}$ and premium $\bar{P}$ functions as

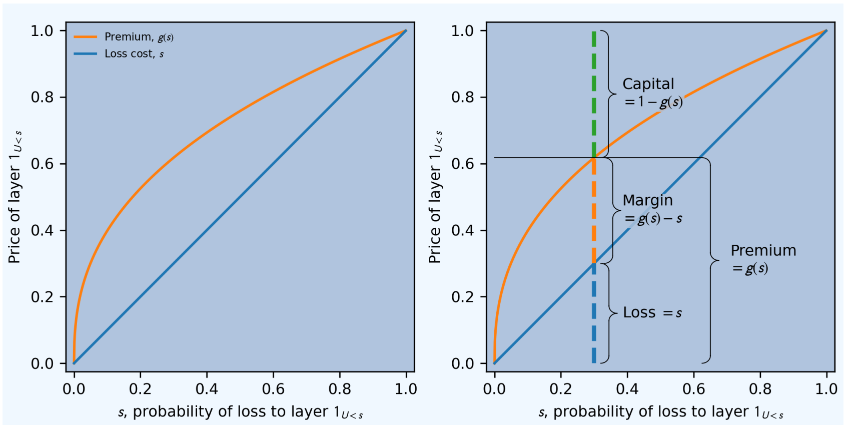
$$\begin{aligned}\bar{S}(a) &= \mathsf{E}[X \wedge a] = \int_0^a S(x)\,dx \\ \bar{P}(a) &= \rho(X \wedge a) = \int_0^\infty g(S_{X\wedge a}(x))\,dx \\ &= \int_0^a g(S_X(x))dx.\end{aligned}$$

Margin is $\bar{M}(a) := \bar{P}(a) - \bar{S}(a)$ and margin density is $M(a) = d\bar{M}(a)/da$. Assets are funded by premium and equity $\bar{Q}(a) := a - \bar{P}(a)$. Again $Q(a) = d\bar{Q}/da = 1 - P(a)$. Together S, M, and Q give the split of layer funding between expected loss, margin and equity. Layers up to a are, by definition, fully collateralized. Thus $\rho(X \wedge a)$ is the premium for a defaultable cover on X supported by assets a, whereas $\rho(X)$ is the premium for an unlimited, default-free cover.

The layer density view is consistent with more standard approaches to pricing. If X is a Bernoulli risk with $\Pr(X = 1) = s$ and expected loss cost s, then $\rho(X) = g(s)$ can be regarded as pricing a unit width layer with attachment probability s. In an intermediated context, the funding constraint requires layers to be fully collateralized by premium plus equity—without such funding the insurance would not be credible since the insurer has no other source of funds.

Given g we can compute insurance market statistics for each layer. The loss, premium, margin, and equity densities are s, $g(s)$, $g(s) - s$ and $1 - g(s)$. The layer loss ratio is $s/g(s)$ and $(g(s) - s)/(1 - g(s))$ is the layer return on equity. These quantities are illustrated in the next figure for a typical distortion function. The corresponding statistics for ground-up covers can be computed by integrating densities.

```
In [1]: from aggregate.extensions.pir_figures import fig_10_3

In [2]: fig_10_3()
```



For an insured risk, we regard the margin as compensation for ambiguity aversion and associated winner's curse drag. Both of these effects are correlated with risk, so the margin is hard to distinguish from a risk load, but the rationale is different. Again, recall, although ρ is non-additive and appears to charge for diversifiable risk, De Waegenaere *et al.* [2003] assures us the pricing is consistent with a general equilibrium.

The layer density is distinct from models that vary the volume of each unit in a homogeneous portfolio model. Our portfolio is static. By varying assets we are implicitly varying the quality of insurance.

5.5.4 Portfolio Pricing with Spectral Risk Measures

Taken as read: a painful discussion that markets set prices, not actuaries and models. *Pricing* here means *valuing* according to some model. For actuaries, valuation is a term of art it means reserving to a life actuary. Pricing actuaries understanding that they are just determining a model value. Thus we will refer to the model price.

Several methods apply a distortion g to price by computing

$$\rho_g(X) = \int g(S(t))dt$$

notably:

1. `Aggregate`: `price`, `apply_distortion`
2. `Portfolio`: `price`, `apply_distortion`, (called by `analyze distortion`)
3. `Distortion`: `price`, `price2`
4. Working by hand using `density_df.p_total`.

All of these methods use the same approach, the integral is approximated as a left Riemann sum:

$$\int_0^\infty g(S(t))dt \approx \sum_{k=0} ng(S(kb))b$$

The implementation computes

- `S` as `1 - p_total.cumsum()`,
- `gS = d.g(S)`, and
- `(gS.loc[:a - bs] * np.diff(S.index)).sum()` or `.cumsum().iloc[-1]`.

The `p_total.cumsum()` idiom automatically accounts for the case where the output distribution is not normalized (sums to < 1). Using `sum` vs. `cumsum` is usually an O(1e-16) difference. These methods all use the forward difference of dt and match against the unlagged values of `S` or `gS` (per PIR p. 272-3). The `Aggregate` method prepends 0 and then computes a `cumsum`, so the `a` index gives the right value. Remember, `pandas.Series.loc[:a]` *includes* the element with index `a` (whereas `iloc[:n]` does not). When `a` is given, the series includes `a` (based on `.loc[:a]`) and the last value is dropped from the sum product.

The next block of code provides a reconciliation of methods. Build an aggregate and put it in a `Portfolio` object to expose `calibrate_distortions`.

```
In [3]: from aggregate import Portfolio, build, qd

In [4]: import pandas as pd

In [5]: a = build('agg CommAuto '
   ...:           '10 claims '
   ...:           '10000 xs 0 '
   ...:           'sev lognorm 50 cv 4 '
   ...:           'poisson')
   ...:

In [6]: qd(a)

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     10                        0.31623            0.31623
Sev   49.804    49.803 -4.6559e-06  3.5917    3.5918   20.434      20.434
Agg   498.04    498.03 -4.6559e-06   1.179     1.179   6.0196      6.0195
log2 = 16, bandwidth = 1/2, validation: not unreasonable.

In [7]: pa = Portfolio('test', [a])
```

(continues on next page)

(continued from previous page)

```
In [8]: pa.update(log2=16, bs=1/4)

In [9]: qd(pa)

                  E[X]  Est E[X]     Err E[X]    CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
unit      X
CommAuto Freq      10                         0.31623            0.31623
         Sev   49.804    49.804 -3.0452e-07  3.5917     3.5918    20.434       20.434
         Agg   498.04    498.04 -1.8636e-06   1.179      1.179    6.0196       6.0168
total    Freq      10                         0.31623            0.31623
         Sev   49.804    49.804 -3.0452e-07  3.5917              20.434
         Agg   498.04    498.03  -1.655e-05   1.179     1.1788    6.0196       6.0109
log2 = 16, bandwidth = 1/4, validation: fails agg mean error >> sev, possible␣
↪aliasing; try larger bs.
```

Determine distortion parameters to achieve a 10% return at 99 percentile capital, and display them. Pull out the achieved pricing.

```
In [10]: pa.calibrate_distortions(ROEs=[0.1], Ps=[0.99], strict='ordered');

In [11]: d = pa.dists['dual']

In [12]: qd(pa.distortion_df)

                              S       L       P       PQ       Q  COC    param       ␣
↪error
a       LR        method                                                             ␣
↪
2.745k 700.875m ccoc   0.0099992 482.03 687.76 0.33431 2057.2  0.1      0.1          ␣
↪  0
                ph     0.0099992 482.03 687.76 0.33431 2057.2  0.1 0.68741   1.
↪7053e-12
                wang   0.0099992 482.03 687.76 0.33431 2057.2  0.1 0.43983    2.
↪558e-11
                dual   0.0099992 482.03 687.76 0.33431 2057.2  0.1  1.9436 -1.
↪1369e-13
                tvar   0.0099992 482.03 687.76 0.33431 2057.2  0.1 0.39096   8.
↪1372e-06

In [13]: f"Exact premium {pa.distortion_df.iloc[0, 2]:.15f}"
Out[13]: 'Exact premium 687.755319984361222'
```

Compute pricing in the four ways described above.

```
In [14]: dm = pa.price(.99, d)

In [15]: f'Exact value {dm.price:.15f}'
Out[15]: 'Exact value 687.755319984360540'

In [16]: bit = a.density_df[['loss', 'p_total', 'S']]

In [17]: bit['aS'] = 1 - bit.p_total.cumsum()

In [18]: bit['gS'] = d.g(bit.S)

In [19]: bit['gaS'] = d.g(bit.aS)

In [20]: test = pd.Series((d.price(bit.loc[:a.q(0.99), 'p_total'], kind='both')[-
↪1],
   ....:                  d.price(a.density_df.p_total, a.q(0.99), kind='both')[-
↪1],
```

(continues on next page)

(continued from previous page)

```
   ....:                    d.price2(bit.p_total).loc[a.q(0.99)].ask, \
   ....:                    d.price2(bit.p_total, a.q(0.99)).ask.iloc[0],
   ....:                    a.price(0.99, d).iloc[0, 1],
   ....:                    dm.price,
   ....:                    bit.loc[:a.q(0.99)-a.bs, 'gS'].sum() * a.bs,
   ....:                    bit.loc[:a.q(0.99)-a.bs, 'gS'].cumsum().iloc[-1] * a.bs,
   ....:                    bit.loc[:a.q(0.99)-a.bs, 'gaS'].sum() * a.bs,
   ....:                    bit.loc[:a.q(0.99)-a.bs, 'gaS'].cumsum().iloc[-1] * a.
↪bs),
   ....:            index=['distortion.price',
   ....:                   'distortion.price with a',
   ....:                   'distortion.price2, find a',
   ....:                   'distortion.price2(a)',
   ....:                   'Aggregate.price',
   ....:                   'Portfolio.price',
   ....:                   'bit sum',
   ....:                   'bit cumsum',
   ....:                   'bit sum alt S',
   ....:                   'bit cumsum alt S'
   ....:                  ])
   ....:
```

Display the results and the relative difference to the largest price.

```
In [21]: qd(test.sort_values(),
   ....:    float_format=lambda x: f'{x:.15f}')
   ....:

distortion.price2(a)         687.755319984360540
distortion.price2, find a    687.755319984360540
Portfolio.price              687.755319984360540
Aggregate.price              687.755319984360540
bit cumsum                   687.755319984360540
bit cumsum alt S             687.755319984360540
distortion.price             687.755319984361108
distortion.price with a      687.755319984361108
bit sum                      687.755319984361108
bit sum alt S                687.755319984361108

In [22]: qd(test.sort_values() / test.sort_values().iloc[-1] - 1,
   ....:    float_format=lambda x: f'{x:.6e}')
   ....:

distortion.price2(a)         -7.771561e-16
distortion.price2, find a    -7.771561e-16
Portfolio.price              -7.771561e-16
Aggregate.price              -7.771561e-16
bit cumsum                   -7.771561e-16
bit cumsum alt S             -7.771561e-16
distortion.price              0.000000e+00
distortion.price with a       0.000000e+00
bit sum                       0.000000e+00
bit sum alt S                 0.000000e+00
```

5.5.5 The Equal Priority Default Rule

If assets are finite and the provider has limited liability we need to to determine policy-level cash flows in default states before we can determine the fair market value of insurance. The most common way to do this is using equal priority in default.

Under limited liability, total losses are split between provider payments and provider default as

$$X = X \wedge a + (X - a)^+.$$

Next, actual payments $X \wedge a$ must be allocated to each policy.

X_i is the amount promised to i by their insurance contract. Promises are limited by policy provisions but are not limited by provider assets. At the policy level, equal priority implies the payments made to, and default borne by, policy i are split as

$$\begin{aligned} X_i &= X_i \frac{X \wedge a}{X} + X_i \frac{(X-a)^+}{X} \\ &= (\text{payments to policy } i) + (\text{default borne by policy } i). \end{aligned}$$

Therefore the payments made to policy i are given

$$X_i(a) := X_i \frac{X \wedge a}{X} = \begin{cases} X_i & X \le a \\ X_i \dfrac{a}{X} & X > a. \end{cases}$$

$X_i(a)$ is the amount actually paid to policy i. It depends on a, X and X_i. The dependence on X is critical. It is responsible for almost all the theoretical complexity of insurance pricing.

It is worth reiterating that with this definition $\sum_i X_i(a) = X \wedge a$.

Example.

Here is an example illustrating the effect of equal priority. Consider a certain loss $X_0 = 1000$ and X_1 given by a lognormal with mean 1000 and coefficient of variation 2.0. Prudence requires losses be backed by assets equal to the 0.9 quantile. On a stand-alone basis X_0 is backed by $a_0 = 1000$ and is risk-free. X_1 is backed by $a_1 = 2272$ and the recovery is subject to a considerable haircut, since $\mathsf{E}[X_1 \wedge 2272] = 732.3$. If these risks are pooled, the pool must hold $a = a_0 + a_1$ for the same level of prudence. When $X_1 \le a_1$ both units are paid in full. But when $X_1 > a_1$, X_0 receives $1000(a/(1000 + X_1))$ and X_1 receives the remaining $X_1(a/(1000 + X_1))$. Payment to both units is pro rated down by the same factor $a/(1000 + X_1)$—hence the name *equal* priority. In the pooled case, the expected recovery to X_0 is 967.5 and 764.8 to X_1. Pooling and equal priority result in a transfer of 32.5 from X_0 to X_1. This example shows what can occur when a thin tailed unit pools with a thick tailed one under a weak capital standard with equal priority. We shall see how pricing compensates for these loss payment transfers, with X_1 paying a positive margin and X_0 a negative one. The calculations are performed in `aggregate` as follows. First, set up the `Portfolio`:

```
In [23]: from aggregate import build, qd

In [24]: port = build('port Dist:EqPri '
   ....:              'agg A 1 claim dsev [1000] fixed '
   ....:              'agg B 1 claim sev lognorm 1000 cv 2 fixed',
   ....:             bs=4)
   ....:

In [25]: qd(port)

              E[X]  Est E[X]     Err E[X]  CV(X)  Est CV(X)  Skew(X)  Est Skew(X)
unit  X
A     Freq      1                              0
      Sev    1000      1000            0       0          0
      Agg    1000      1000            0       0          0
B     Freq      1                              0
      Sev    1000    999.91  -8.6294e-05       2     1.9921       14       12.417
```

(continues on next page)

(continued from previous page)

```
      Agg   1000   999.91 -8.6294e-05       2    1.9921       14       12.417
total Freq     2                            0
      Sev   1000   999.96 -4.3147e-05 1.4142            19.799
      Agg   2000   1999.9 -4.3673e-05       1   0.99599       14        12.41
log2 = 16, bandwidth = 4, validation: fails sev cv, agg cv.
```

`var_dict()` returns the 90th percentile points by unit and in total.

```
In [26]: port.var_dict(.9)
Out[26]: {'A': 1000.0, 'B': 2272.0, 'total': 3272.0}
```

Extract the relevant fields from `density_df` for the allocated loss recoveries. The first block shows standalone, the second pooled.

```
In [27]: qd(port.density_df.filter(regex='S|lev_[ABt]').loc[[port.B.q(0.9)]])

             S  lev_total   lev_A   lev_B
2.272k 0.20463       1589    1000 732.35

In [28]: qd(port.density_df.filter(regex='S|exa_[ABt]').loc[[port.q(0.9)]])

              S  exa_total   exa_A   exa_B
3.272k 0.099939     1732.4 967.51 764.85
```

5.5.6 Expected Loss Payments at Different Asset Levels

Expected losses paid to policy i are $\bar S_i(a) := \mathsf{E}[X_i(a)]$. $\bar S_i(a)$ can be computed, conditioning on X, as

$$\bar S_i(a) = \mathsf{E}[\mathsf{E}[X_i(a) \mid X]] = \mathsf{E}[X_i \mid X \le a]F(a) + a\mathsf{E}\left[\frac{X_i}{X} \mid X > a\right] S(a).$$

Because of its importance in allocating losses, define

$$\alpha_i(a) := \mathsf{E}[X_i/X \mid X > a].$$

The value $\alpha_i(x)$ is the expected proportion of recoveries by unit i in the layer at x. Since total assets available to pay losses always equals the layer width, and the chance the layer attaches is $S(x)$, it is intuitively clear $\alpha_i(x)S(x)$ is the loss density for unit i, that is, the derivative of $\bar S_i(x)$ with respect to x. We now show this rigorously.

Proposition. Expected losses to policy i under equal priority, when total losses are supported by assets a, is given by

$$\bar S_i(a) = \mathsf{E}[X_i(a)] = \int_0^a \alpha_i(x)S(x)dx$$

and so the policy loss density at x is $S_i(x) := \alpha_i(x)S(x)$.

Proof. By the definition of conditional expectation, $\alpha_i(a)S(a) = \mathsf{E}[(X_i/X)1_{X>a}]$. Conditioning on X, using the tower property, and taking out the functions of X on the right shows

$$\alpha_i(a)S(a) = \mathsf{E}[\mathsf{E}[(X_i/X)1_{X>a} \mid X]] = \int_a^\infty \mathsf{E}[X_i \mid X = x]\frac{f(x)}{x}dx$$

and therefore

$$\frac{d}{da}(\alpha_i(a)S(a)) = -\mathsf{E}[X_i \mid X = a]\frac{f(a)}{a}.$$

Now we can use integration by parts to compute

$$\begin{aligned}\int_0^a \alpha_i(x)S(x)\,dx &= x\alpha_i(x)S(x)\Big|_0^a + \int_0^a x\,\mathsf{E}[X_i \mid X = x]\frac{f(x)}{x}\,dx \\ &= a\alpha_i(a)S(a) + E[X_i \mid X \le a]F(a) \\ &= \bar S_i(a).\end{aligned}$$

Therefore the policy i loss density in the asset layer at a, i.e. the derivative of cref{eq:eloss-main} with respect to a, is $S_i(a) = \alpha_i(a)S(a)$ as required.

Note that S_i is *not* the survival function of $X_i(a)$ nor of X_i.

5.5.7 The Natural Allocation Premium

Premium under ρ is given by $\int_0^a g(S)$. We can interpret $g(S(a))$ as the portfolio premium density in the layer at a. We now consider the premium and premium density for each policy.

Using integration by parts we can express the price of an unlimited cover on X as

$$\rho(X) = \int_0^\infty g(S(x))\,dx = \int_0^\infty xg'(S(x))f(x)\,dx = \mathsf{E}[Xg'(S(X)))].$$

It is important that this integral is over all $x \ge 0$ so the $xg(S(x))|_0^a$ term disappears. The formula makes sense because a concave distortion is continuous on $(0,1]$ and can have at most countably infinitely many points where it is not differentiable (it has a kink). In total these points have measure zero, Borwein and Vanderwerff [2010], and we can ignore them in the integral. For more details see Dhaene *et al.* [2012].

Combining the integral and the properties of a distortion function, $g'(S(X))$ is the Radon-Nikodym derivative of a measure Q with $\rho(X) = \mathsf{E}_{\mathsf{Q}}[X]$. In fact, $\mathsf{E}_{\mathsf{Q}}[Y] = \mathsf{E}[Yg'(S(X))]$ for all random variables Y. In general, any non-negative function Z (measure Q) with $\mathsf{E}[Z] = 1$ and $\rho(X) = \mathsf{E}[XZ]$ $(= \mathsf{E}_{\mathsf{Q}}[X])$ is called a contact function (subgradient) for ρ at X, see Shapiro *et al.* [2009]. Thus $g'(S(X))$ is a contact function for ρ at X. The name subgradient comes from the fact that $\rho(X+Y) \ge \mathsf{E}_{\mathsf{Q}}[X+Y] = \rho(X) + \mathsf{E}_{\mathsf{Q}}[Y]$, by the representation theorem. The set of subgradients is called the subdifferential of ρ at X. If there is a unique subgradient then ρ is differentiable. Delbaen [2000] Theorem 17 shows that subgradients are contact functions.

We can interpret $g'(S(X))$ as a state price density specific to the X, suggesting that $\mathsf{E}[X_i g'(S(X))]$ gives the value of the cash flows to policy i. This motivates the following definition.

Definition. For $X = \sum_i X_i$ with $\mathsf{Q} \in \mathcal{Q}$ so that $\rho(X) = \mathsf{E}_{\mathsf{Q}}[X]$, the **natural allocation premium** to policy X_j as part of the portfolio X is $\mathsf{E}_{\mathsf{Q}}[X_j]$. It is denoted $\rho_X(X_j)$.

The natural allocation premium is a standard approach, appearing in Delbaen [2000], Venter *et al.* [2006] and Tsanakas and Barnett [2003] for example. It has many desirable properties. Delbaen shows it is a fair allocation in the sense of fuzzy games and that it has a directional derivative, marginal interpretation when ρ is differentiable. It is consistent with Jouini and Kallal [2001] and Campi *et al.* [2013], which show the rational price of X in a market with frictions must be computed by state prices that are anti-comonotonic X. In our application the signs are reversed: $g'(S(X))$ and X are comonotonic.

The choice $g'(S(X))$ is economically meaningful because it weights the largest outcomes of X the most, which is appropriate from a social, regulatory and investor perspective. It is also the only choice of weights that works for all levels of assets. Since investors stand ready to write any layer at the price determined by g, their solution must work for all a.

However, there are two technical issues with the proposed natural allocation. First, unlike prior works, we are allocating the premium for $X \wedge a$, not X, a problem also considered in Major [2018]. And second, Q may not be unique. In general, uniqueness fails at capped variables like $X \wedge a$. Both issues are surmountable for a SRM, resulting in a unique, well defined natural allocation. For a non-comonotonic additive risk measure this is not the case.

It is helpful to define the premium, risk adjusted, analog of the α_i as

$$\beta_i(a) := \mathsf{E}_{\mathsf{Q}}[(X_i/X) \mid X > a].$$

$\beta_i(x)$ is the value of the recoveries paid to unit i by a policy paying 1 in states $\{X > a\}$, i.e. an allocation of the premium for $1_{X>a}$. By the properties of conditional expectations, we have

$$\beta_i(a) = \frac{\mathsf{E}[(X_i/X)Z \mid X > a]}{\mathsf{E}[Z \mid X > a]}.$$

The denominator equals $\mathsf{Q}(X > a)/\mathsf{P}(X > a)$. Remember that while $\mathsf{E}_{\mathsf{Q}}[X] = \mathsf{E}[XZ]$, for conditional expectations $\mathsf{E}_{\mathsf{Q}}[X \mid \mathcal{F}] = \mathsf{E}[XZ \mid \mathcal{F}]/\mathsf{E}[Z \mid \mathcal{F}]$, see [Föllmer and Schied [2011], Proposition A.12].

To compute α_i and β_i we use a third function,

$$\kappa_i(x) := \mathsf{E}[X_i \mid X = x],$$

the conditional expectation of loss by policy, given the total loss.

Theorem. Let $\mathsf{Q} \in \mathcal{Q}$ be the measure with Radon-Nikodym derivative $Z = g'(S_X(X))$. Then:

1. $\mathsf{E}[X_i \mid X = x] = \mathsf{E}_{\mathsf{Q}}[X_i \mid X = x]$.
2. β_i can be computed from κ_i as

$$\beta_i(a) = \frac{1}{\mathsf{Q}(X > a)} \int_a^\infty \frac{\kappa_i(x)}{x} g'(S(x))f(x)\,dx.$$

1. The natural allocation premium for policy i under equal priority when total losses are supported by assets a, $\bar{P}_i(a) := \rho_{X\wedge a}(X_i(a))$, is given by

$$\begin{aligned}\bar{P}_i(a) &= \mathsf{E}_{\mathsf{Q}}[X_i \mid X \le a](1 - g(S(a))) + a\mathsf{E}_{\mathsf{Q}}[X_i/X \mid X > a]g(S(a)) \\ &= \mathsf{E}[X_i Z \mid X \le a](1 - S(a)) + a\mathsf{E}[(X_i/X)Z \mid X > a]S(a).\end{aligned}$$

1. The policy i premium density equals

$$P_i(a) = \beta_i(a)g(S(a)).$$

It is an important to know when the natural allocation premium is unique. It is so when Z is the only contact function (i.e., there are no others). If X has a strictly increasing quantile function or is injective then Q is unique and therefore given by $g'S(X)$ and hence X measurable, see [Carlier and Dana, 2003] and Marinacci and Montrucchio [2004]. More generally, we can replace Q with its expectation given X to make a canonical choice, resulting in the linear natural allocation [Cherny and Orlov, 2011].

The problem that can occur when Q is not unique, but that can be circumvented when ρ is a SRM, can be illustrated as follows. Suppose ρ is given by p-TVaR. The measure Q weights the worst $1 - p$ proportion of outcomes of X by a factor of $(1 - p)^{-1}$ and ignores the others. Suppose a is chosen as p'-VaR for a lower threshold $p' < p$. Let $X_a = X \wedge a$ be capped insured losses and $C = \{X_a = a\}$. By definition $\Pr(C) \ge 1 - p' > 1 - p$. Pick any $A \subset C$ of measure $1 - p$ so that $\rho(X) = \mathsf{E}[X \mid A]$. Let ψ be a measure preserving transformation of Ω that acts non-trivially on C but trivially off C. Then $\mathsf{Q}' = \mathsf{Q}\psi$ will satisfy $\mathsf{E}_{\mathsf{Q}'}[X_a] = \mathsf{E}_{\mathsf{Q}}[X_a\psi^{-1}] = \rho(X_a)$ but in general $\mathsf{E}_{\mathsf{Q}'}[X] < \rho(X)$. The natural allocation with respect to Q' will be different from that for Q. The theorem isolates a specific Q to obtain a unique answer. The same idea applies to Q from other, non-TVaR, ρ: you can always shuffle part of the contact function within C to generate non-unique allocations. See Mildenhall and Major [2022] Example 239 for an illustration.

When Q is X measurable, then $\mathsf{E}_{\mathsf{Q}}[X_i \mid X] = \mathsf{E}[X_i \mid X]$, which enables explicit calculation. In this case there is no risk adjusted version of κ_i. If Q is not X measurable, then there can be risk adjusted κ_i because

$$\mathsf{E}[X_i Z \mid X] \ne \mathsf{E}[X_i \mid X]\mathsf{E}[Z \mid X].$$

The proof writes the price of a limited liability cover as the price of default-free protection minus the value of the default put. This is the standard starting point for allocation in a perfect competitive market taken by Phillips *et al.* [1998], Myers and Read Jr. [2001], Sherris [2006], and Ibragimov *et al.* [2010]. They then allocate the default put rather than the value of insurance payments directly.

To recap: the premium formulas have been derived assuming capital is provided at a cost g and there is equal priority by unit. The formulas are computationally tractable (see implementation in 5_x_portfolio_calculations) and require only that X have an increasing quantile function or that $g'S(X)$ be used as the risk adjustment, but make no other assumptions. There is no need to assume the X_i are independent. They produce an entirely general, canonical determination of premium in the presence of shared costly capital. This result extends Grundl and Schmeiser [2007], who pointed out that with an additive pricing functional there is no need to allocate capital in order to price, to the situation of a non-additive SRM pricing functional.

5.5.8 Properties of Alpha, Beta, and Kappa

In this section we explore properties of α_i, β_i, and κ_i, and show how they interact to determine premiums by unit via the natural allocation.

For a measurable h, $\mathsf{E}[X_i h(X)] = \mathsf{E}[\kappa_i(X)h(X)]$ by the tower property. This simple observation results in huge simplifications. In general, $\mathsf{E}[X_i h(X)]$ requires knowing the full bivariate distribution of X_i and X. Using κ_i reduces it to a one dimensional problem. This is true even if the X_i are correlated. The κ_i functions can be estimated from data using regression and they provide an alternative way to model correlations.

Despite their central role, the κ_i functions are probably unfamiliar so we begin by giving several examples to illustrate how they behave. In general, they are non-linear and usually, but not always, increasing.

Examples of κ functions

1. If Y_i are independent and identically distributed and $X_n = Y_1 + \cdots + Y_n$ then $\mathsf{E}[X_m \mid X_{m+n} = x] = mx/(m+n)$ for $m \ge 1, n \ge 0$. This is obvious when $m = 1$ because the functions $\mathsf{E}[Y_i \mid X_n]$ are independent across $i = 1, \dots, n$ and sum to x. The result follows because conditional expectations are linear. In this case $\kappa_i(x) = mx/(m+n)$ is a line through the origin.
2. If X_i are multivariate normal then κ_i are straight lines, given by the usual least-squares fits

$$\kappa_i(x) = \mathsf{E}[X_i] + \frac{\mathsf{cov}(X_i, X)}{\mathsf{var}(X)}(x - \mathsf{E}[X]).$$

   This example is familiar from the securities market line and the CAPM analysis of stock returns. If X_i are iid it reduces to the previous example because the slope is $1/n$.
3. If X_i, $i = 1, 2$, are compound Poisson with the same severity distribution then κ_i are again lines through the origin. Suppose X_i has expected claim count λ_i. Write the conditional expectation as an integral, expand the density of the compound Poisson by conditioning on the claim count, and then swap the sum and integral to see that $\kappa_1(x) = \mathsf{E}[X_1 \mid X_1 + X_2 = x] = x\,\mathsf{E}[N(\lambda_1)/(N(\lambda_1) + N(\lambda_2))]$ where $N(\lambda)$ are independent Poisson with mean λ. This example generalizes the iid case. Further conditioning on a common mixing variable extends the result to mixed Poisson frequencies where each aggregate can have a separate or shared mixing distribution. The common severity is essential. The result means that if a line of business is defined to be a group of policies that shares the same severity distribution, then premiums for policies within the line will have rates proportional to their expected claim counts.
4. A theorem of Efron says that if X_i are independent and have log-concave densities then all κ_i are non-decreasing, Saumard and Wellner [2014]. The multivariate normal example is a special case of Efron's theorem.

Denuit and Dhaene [2012] define an ex post risk sharing rule called the conditional mean risk allocation by taking $\kappa_i(x)$ to be the allocation to policy i when $X = x$. A series of recent papers, see Denuit and Robert [2020] and references therein, considers the properties of the conditional mean risk allocation focusing on its use in peer-to-peer insurance and the case when $\kappa_i(x)$ is linear in x.

5.5.9 Properties of the Natural Allocation

We now explore margin, equity, and return in total and by policy. We begin by considering them in total.

By definition the average return with assets a is

$$\bar\iota(a) := \frac{\bar M(a)}{\bar Q(a)}$$

where margin $\bar M$ and equity $\bar Q$ are the total margin and capital functions defined above.

The last formula has important implications. It tells us the investor priced expected return varies with the level of assets. For most distortions return decreases with increasing capital. In contrast, the standard RAROC models use a fixed average cost of capital, regardless of the overall asset level, Tasche [1999]. CAPM or the Fama-French three factor model are often used to estimate the average return, with a typical range of 7 to 20 percent, Cummins and Phillips [2005]. A common question of working actuaries performing capital allocation is about so-called excess capital, if the balance sheet contains more capital than is required by regulators, rating agencies, or managerial prudence. Our model suggests that higher layers of capital are cheaper, but not free, addressing this concern.

The varying returns may seem inconsistent with Miller-Modigliani. But that says the cost of funding a given amount of capital is independent of how it is split between debt and equity; it does not say the average cost is constant as the amount of capital varies.

No-Undercut and Positive Margin for Independent Risks

The natural allocation has two desirable properties. It is always less than the stand-alone premium, meaning it satisfies the no-undercut condition of Denault [2001], and it produces non-negative margins for independent risks.

Proposition. Let $X = \sum_{i=1}^n X_i$, X_i non-negative and independent, and let g be a distortion. Then

1. the natural allocation is never greater than the stand-alone premium, and
2. the natural allocation to every X_i contains a non-negative margin.

Since $\bar{P}_i = \mathsf{E}[\kappa_i(X)g'(S(X))]$ we see the no-undercut condition holds if $\kappa_i(X)$ and $g'(S(X))$ are comonotonic, and hence if κ_i is increasing, or if $\kappa_i(X)$ and X are positively correlated (recall $\mathsf{E}[g'(S(X))] = 1$). A policy i^* with increasing κ_{i^*} is a capacity consuming line that always has a positive margin. However, it can occur that no κ_i is increasing.

Policy Level Properties, Varying with Asset Level

We start with a corollary which gives a nicely symmetric and computationally tractable expression for the natural margin allocation in the case of finite assets.

Corollary. The margin density for unit i at asset level a is given by

$$M_i(a) = \beta_i(a)g(S(a)) - \alpha_i(a)S(a).$$

Proof. We can compute margin $\bar{M}_i(a)$ in $\bar{P}_i(a)$ by line as

$$\begin{aligned}\bar{M}_i(a) =& \bar{P}_i(a) - \bar{L}_i(a) \\ =& \int_0^a \beta_i(x)g(S(x)) - \alpha_i(x)S(x)\,dx.\end{aligned}$$

Differentiating we get the margin density for unit i at a expressed in terms of α_i and β_i as shown.

Margin in the current context is the cost of capital, thus this is an important result. It allows us to compute economic value by unit and to assess static portfolio performance by unit—one of the motivations for performing capital allocation in the first place. In many ways it is also a good place to stop. Remember these results only assume we are using a distortion risk measure and have equal priority in default. We are in a static model, so questions of portfolio homogeneity are irrelevant. We are not assuming X_i are independent.

What can we say about by margins by unit? Since g is increasing and concave $P(a) = g(S(a)) \ge S(a)$ for all $a \ge 0$. Thus all asset layers contain a non-negative total margin density. It is a different situation by unit, where we can see

$$M_i(a) \ge 0 \iff \beta_i(a)g(S(a)) - \alpha_i(a)S(a) \ge 0 \iff \frac{\beta_i(a)}{\alpha_i(a)} \ge \frac{S(a)}{g(S(a))}.$$

The unit layer margin density is positive when β_i/α_i is greater than the all-unit layer loss ratio. Since the loss ratio is ≤ 1 there must be a positive layer margin density whenever $\beta_i(a)/\alpha_i(a) > 1$. But when $\beta_i(a)/\alpha_i(a) < 1$ it is possible the unit has a negative margin density. How can that occur and why does it make sense? To explore this we look at the shape of α and β in more detail.

It is important to remember why the Proposition does not apply: it assumes unlimited cover, whereas here $a < \infty$. With finite capital there are potential transfers between units caused by their behavior in default that overwhelm the positive margin implied by the proposition. Also note the proposition cannot be applied to $X \wedge a = \sum_i X_i(a)$ because the unit payments are no longer independent.

In general we can make two predictions about margins.

Prediction 1: Lines where $\alpha_i(x)$ or $\kappa_i(x)/x$ increase with x will have always have a positive margin.

Prediction 2: A log-concave (thin tailed) unit aggregated with a non-log-concave (thick tailed) unit can have a negative margin, especially for lower asset layers.

Prediction 1 follows because the risk adjustment puts more weight on X_i/X for larger X and so $\beta_i(x)/\alpha_i(x) > 1 > S(x)/g(S(x))$. Recall the risk adjustment is comonotonic with total losses X.

A thin tailed unit aggregated with thick tailed units will have $\alpha_i(x)$ decreasing with x. Now the risk adjustment will produce $\beta_i(x) < \alpha_i(x)$ and it is possible that $\beta_i(x)/\alpha_i(x) < S(x)/g(S(x))$. In most cases, $\alpha_i(x)$ approaches $\mathsf{E}[X_i]/x$ and $\beta_i(x)/\alpha_i(x)$ increases with x, while the layer loss ratio decreases—and margin increases—and the thin unit will eventually get a positive margin. Whether or not the thin unit has a positive total margin $\bar{M}_i(a) > 0$ depends on the particulars of the units and the level of assets a. A negative margin is more likely for less well capitalized insurers, which makes sense because default states are more material and they have a lower overall dollar cost of capital. In the independent case, as $a \to \infty$ the proposition guarantees an eventually positive margins for all units.

These results are reasonable. Under limited liability, if assets and liabilities are pooled then the thick tailed unit benefits from pooling with the thin one because pooling increases the assets available to pay losses when needed. Equal priority transfers wealth from thin to thick in states of the world where thick has a bad event. But because thick dominates the total, the total losses are bad when thick is bad. The negative margin compensates the thin-tailed unit for transfers.

Another interesting situation occurs for asset levels within attritional loss layers. Most realistic insured loss portfolios are quite skewed and never experience very low loss ratios. For low loss layers, $S(x)$ is close to 1 and the layer at x is funded almost entirely by expected losses; the margin and equity density components are nearly zero. Since the sum of margin densities over component units equals the total margin density, when the total is zero it necessarily follows that either all unit margins are also zero or that some are positive and some are negative. For the reasons noted above, thin tailed units get the negative margin as thick tailed units compensate them for the improved cover the thick tail units obtain by pooling.

In conclusion, the natural margin by unit reflects the relative consumption of assets by layer, Mango [2005]. Low layers are less ambiguous to the provider and have a lower margin relative to expected loss. Higher layers are more ambiguous and have lower loss ratios. High risk units consume more higher layer assets and hence have a lower loss ratio. For independent units with no default the margin is always positive. But there is a confounding effect when default is possible. Because more volatile units are more likely to cause default, there is a wealth transfer to them. The natural premium allocation compensates low risk policies for this transfer, which can result in negative margins in some cases.

5.5.10 The Natural Allocation of Equity

Although we have a margin by unit, we cannot compute return by unit, or allocate frictional costs of capital, because we still lack an equity allocation, a problem we now address.

Definition. The **natural allocation of equity** to unit i is given by

$$Q_i(a) = \frac{\beta_i(a)g(S(a)) - \alpha_i(x)S(a)}{g(S(a)) - S(a)} \times (1 - g(S(a))).$$

Why is this allocation natural? In total the layer return at a is

$$\iota(a) := \frac{M(a)}{Q(a)} = \frac{P(a) - S(a)}{1 - P(a)} = \frac{g(S(a)) - S(a)}{1 - g(S(a))}.$$

We claim that for a law invariant pricing measure the layer return *must be the same for all units*. Law invariance implies the risk measure is only concerned with the attachment probability of the layer at a, and not with the cause of loss within the layer. If return *within a layer* varied by unit then the risk measure could not be law invariant.

We can now compute capital by layer by unit, by solving for the unknown equity density $Q_i(a)$ via

$$\iota(a) = \frac{M(a)}{Q(a)} = \frac{M_i(a)}{Q_i(a)} \implies Q_i(a) = \frac{M_i(a)}{\iota(a)}.$$

Substituting for layer return and unit margin gives the result.

Since $1 - g(S(a))$ is the proportion of capital in the layer at a, the main allocation result says the allocation to unit i is given by the nicely symmetric expression

$$\frac{\beta_i(a)g(S(a)) - \alpha_i(x)S(a)}{g(S(a)) - S(a)}.$$

To determine total capital by unit we integrate the equity density

$$\bar{Q}_i(a) := \int_0^a Q_i(x)dx.$$

And finally we can determine the average return to unit i at asset level a

$$\bar{\iota}_i(a) = \frac{\bar{M}_i(a)}{\bar{Q}_i(a)}.$$

The average return will generally vary by unit and by asset level a. Although the return within each layer is the same for all units, the margin, the proportion of capital, and the proportion attributable to each unit all vary by a. Therefore average returns will vary by unit and a. This is in stark contrast to the standard industry approach, which uses the same return for each unit and implicitly all a. How these quantities vary by unit is complicated. Academic approaches emphasized the possibility that returns vary by unit, but struggled with parameterization, Myers and Cohn [1987].

This formula shows the average return by unit is an M_i-weighted harmonic mean of the layer returns given by the distortion g, viz

$$\frac{1}{\bar{\iota}_i(a)} = \int_0^a \frac{1}{\iota(x)} \frac{M_i(x)}{\bar{M}_i(a)} dx.$$

The harmonic mean solves the problem that the return for lower layers of assets is potentially infinite (when $g'(1) = 0$). The infinities do not matter: at lower asset layers there is little or no equity and the layer is fully funded by the loss component of premium. When so funded, there is no margin and so the infinite return gets zero weight. In this instance, the sense of the problem dictates that $0 \times \infty = 0$: with no initial capital there is no final capital regardless of the return.

5.5.11 Appendix: Notation and Conventions

An insurer has finite assets and limited liability and is a one-period stock company. At $t = 0$ it sells its residual value to investors to raise equity. At time one it pays claims up to the amount of assets available. If assets are insufficient to pay claims it defaults. If there are excess assets they are returned to investors.

Total insured loss, or total risk, is described by a random variable $X \geq 0$. X reflects policy limits but is not limited by provider assets. $X = \sum_i X_i$ describes the split of losses by policy. F, S, f, and q are the distribution, survival, density, and (lower) quantile functions of X. Subscripts are used to disambiguate, e.g., S_{X_i} is the survival function of X_i. $X \wedge a$ denotes $\min(X, a)$ and $X^+ = \max(X, 0)$.

The letters S, P, M and Q refer to expected loss, premium, margin and equity, and a refers to assets. The value of survival function $S(x)$ is the loss cost of the insurance paying $1_{\{X>x\}}$, so the two uses of S are consistent. Premium equals expected loss plus margin; assets equal premium plus equity. All these quantities are functions of assets underlying the insurance.

We use the actuarial sign convention: large positive values are bad. Our concern is with quantiles $q(p)$ for p near 1. Distortions are usually reversed, with $g(s)$ for small $s = 1 - p$ corresponding to bad outcomes. As far as possible we will use p in the context p close to 1 is bad and s when small s is bad.

Tail value at risk is defined for $0 \leq p < 1$ by

$$\mathsf{TVaR}_p(X) = \frac{1}{1-p} \int_p^1 q(t)dt.$$

Prices exclude all expenses. The risk free interest rate is zero. These are standard simplifying assumptions, e.g. Ibragimov *et al.* [2010].

The terminology describing risk measures is standard, and follows Föllmer and Schied [2011]. We work on a standard probability space, Svindland [2010], Appendix. It can be taken as $\Omega = [0, 1]$, with the Borel sigma-algebra and P Lebesgue measure. The indicator function on a set A is 1_A, meaning $1_A(x) = 1$ if $x \in A$ and $1_A(x) = 0$ otherwise.

5.6 Bodoff's Percentile Layer Capital Method

Objectives: Compare Bodoff with the natural allocation and show how to compute both in `aggregate`.

Audience: Those interested in current allocation methods and CAS Exam 9 candidates.

Prerequisites: Background on allocation and Bodoff's paper.

Contents:

5.6.1 Helpful References

- Bodoff [2007]
- Mildenhall and Major [2022]

5.6.2 Introduction

The abstract to Bodoff [2007], Capital Allocation by Percentile Layer reads:

> This paper describes a new approach to capital allocation; the catalyst for this new approach is a new formulation of the meaning of holding Value at Risk (VaR) capital. This new formulation expresses the firm's total capital as the sum of many granular pieces of capital, or "percentile layers of capital." As a result, one must allocate capital separately on each layer and perform the capital allocation across all layers. The resulting capital allocation procedure, "capital allocation by percentile layer," exhibits several salient features. First, **it allocates capital to all losses, rather than allocating capital only to extreme losses in the tail of the distribution**. Second, despite allocating capital to this broad range of loss events, **the proposed procedure does not allocate in proportion to average loss; rather, it allocates disproportionate capital to severe losses**. Third, **it allocates capital by relying neither upon esoteric parameters nor upon elusive risk preferences**. Ultimately, on the practical plane, **capital allocation by percentile layer produces allocations that are different from many other methods**. Concomitantly, on the theoretical plane, capital allocation by percentile layer leads to new continuous formulas for risk load and utility.

Bodoff's paper is an important contribution to capital allocation and actuarial science. Its key insight is that layers of capital respond to a range of loss events and not just tail events and so it is not appropriate to focus solely on default states when allocating capital. Bodoff takes capital to mean total claims paying ability, comprised of equity and premium. Bodoff allocates capital by considering loss outcomes and assumes that expected loss, margin, premium, and equity all have the same allocation **within each layer**.

Less favorably, Bodoff blurs the distinction between events and outcomes. He allocates to identifiable **events** (wind-only loss, etc.) rather than to **outcomes**. In examples, outcome amounts distinguish events. In the Lee diagram, events are on the horizontal axis and outcomes on the vertical axis.

5.6.3 Assumptions and Notation

The examples model two independent units X_1 and X_2, usually `wind` and `quake`, with total $X = X_1 + X_2$. F and S represent the distribution and survival function of X and q its lower quantile function. The capital (asset) requirement set equal to the (lower) $a := p = 0.99$-VaR capital

5.6.4 Three Possible Allocation Methods

Consider three allocations:

1. **Conditional VaR**: `coVaR`, method allocates using

$$a = \mathsf{E}[X \mid X = a] = \mathsf{E}[X_1 \mid X = a] + \mathsf{E}[X_2 \mid X = a]$$

2. **Alternative conditional VaR**: `alt coVaR`, method allocates using

$$a = a\,\mathsf{E}\left[\frac{X_1}{X} \mid X \ge a\right] + a\,\mathsf{E}\left[\frac{X_2}{X} \mid X \ge a\right]$$

3. **Naive conditional TVaR**: `naive coTVaR`, method allocates a proportional to $\mathsf{E}[X_1 \mid X \ge a]$ and $\mathsf{E}[X_2 \mid X \ge a]$

Bodoff's principal criticism of these methods is that they all ignore the possibility of outcomes $< a$.

- `coVaR` allocates based proportion of losses by unit on the events $\{X = a\}$ of exact size a. It ignores other events near $X = a$ and all events $X < a$, which seems unreasonable. The allocation is not numerically stable: in simulation output $\{X = a\}$ is often only a single event.
- `alt coVaR` allocates based proportion of losses by unit on the events $\{X \ge a\}$. It still ignores all events $< a$. It relies on the relationship

$$\begin{aligned} a &= a\left(\mathsf{E}\left[\frac{X_1}{X} \mid X \ge a\right] + a\mathsf{E}\left[\frac{X_2}{X} \mid X \ge a\right]\right) \\ &= a\,\alpha_1(a) + a\,\alpha_2(a) \end{aligned}$$

- `naive coTVaR` resorts to a pro rata kludge because $\mathsf{E}[X \mid X \ge x] \ge x$ and is usually $> x$. Pro rata adjustments signal the lack of a rigorous rationale and should be avoided. Note: what Bodoff calls TVaR is usually known as CTE.
- **Alternative conditional TVaR**: the `coTVaR` method (not considered by Bodoff but introduced by Mango, Venter, Kreps, Major) solves $a = \mathsf{TVaR}(p^*)$ for $p^* \le p$ (we shall see below we really need to use expected shortfall, not TVaR). Then determine $a^* = q(p^*)$, the p^*-VaR and allocate using $a = \mathsf{E}[X \mid X \ge a^*] = \mathsf{E}[X_1 \mid X \ge a^*] + \mathsf{E}[X_2 \mid X \ge a^*]$.

In addition, all methods can be criticized as actuarial allocation exercises without an economic motivation. They do not consider premium: additional assumptions needed to derive a premium from an asset or capital allocation, such as a target return on allocated capital. They just provide an allocation of premium plus capital, i.e., assets, and not a split between the two.

5.6.5 Percentile Layer Allocation: Definition

Bodoff introduces the **percentile layer of capital**, `plc`, allocation method to address the criticism that methods 1-4 all ignore events causing losses below the level of capital, whereas capital is certainly used to pay such losses. It allocates capital in the same proportion as losses for each layer.

In a one-dollar, all-or-nothing cover that attaches with probability $s = 1 - p$ at $x = q(p)$ ($= p$-VaR), under equal priority unit i receives a proportion $\alpha_i(x) := \mathsf{E}\left[\dfrac{X_i}{X} \mid X > x\right]$ of assets, conditional on a loss. Therefore, unconditional expected loss recoveries equal $\alpha_i(x)S(x)$, part of total layer losses $S(x)$. Allocating each layer of capital between 0 and a in the same way gives the **percentile layer of capital** `plc` allocation:

$$a_i := \int_0^a \alpha_i(x)\,dx = \int_0^a \mathsf{E}\left[\frac{X_i}{X} \mid X > x\right]\,dx$$

By construction, $\sum_i a_i = a$. The `plc` allocation can be understood better by decomposing

$$\begin{aligned}
a &= \int_0^a 1\,dx \\
&= \int_0^a \alpha_1(x) + \alpha_2(x)\,dx \\
&= \int_0^a \alpha_1(x)S(x) + \alpha_1(x)F(x)\,dx + \int_0^a \alpha_2(x)S(x) + \alpha_2(x)F(x)\,dx \\
&= \left(\mathsf{E}[X_1(a)] + \int_0^a \alpha_1(x)F(x)\,dx\right) + \left(\mathsf{E}[X_2(a)] + \int_0^a \alpha_2(x)F(x)\,dx\right)
\end{aligned}$$

It splits unfunded assets (assets in excess of expected losses) in the same proportion as losses in each asset layer, using $\alpha_i(x)$. `plc` says **nothing** about how to split the allocated unfunded capital $\int_0^a \alpha_2(x)F(x)\,dx$ into margin and equity. This is not surprising, since there are no pricing assumptions. The natural allocation introduces a pricing distortion to compute an allocation of premium, and hence margin.

There are six allocations considered by Bodoff, with the following allocations of assets to unit 1.

1. `pct EX`: $\mathsf{E}[X_1]/\mathsf{E}[X]$
2. `coVaR`: $\mathsf{E}[X_1 \mid X = a]$
3. `adj VaR`: $a\,\mathsf{E}\left[\dfrac{X_1}{X} \mid X \ge a\right]$
4. `naive coTVaR`: $a\,\dfrac{\mathsf{E}[X_1 \mid X \ge a]}{\mathsf{E}[X \mid X \ge a]}$
5. `coTVaR`: $\mathsf{E}[X_1 \mid X > a^*]$, where $a = \mathsf{TVaR}(p^*)$
6. `plc`: $\displaystyle\int_0^a \alpha_i(x)\,dx$, where $\alpha_i(x) := \mathsf{E}\left[\dfrac{X_i}{X} \mid X > x\right]$

5.6.6 Thought Experiments

Bodoff introduces four thought experiments:

1. Wind and quake, wind losses 0 or 99, quake 0 or 100, 0.2 probability of a wind loss and 0.01 probability of a quake loss.
2. Wind and quake, wind 0 or 50, quake 0 or 100, same probabilities.
3. Wind and quake, wind 0 or 5, quake 0 or 100, same probabilities.
4. Bernoulli / exponential compound distribution (see *Bodoff Example 4*.)

The units are independent. The next block of code sets up and validates `Portfolio` objects for each. The Bodoff portfolios are part of the base library and can be extracted with `build.qlist`.

```
In [1]: import pandas as pd

In [2]: from collections import OrderedDict

In [3]: from aggregate import build, qd

In [4]: from aggregate.extensions import bodoff_exhibit

In [5]: bodoff = list(build.qlist('.*Bodoff').program)

In [6]: ports = OrderedDict()

In [7]: for s in bodoff:
   ...:     port = build(s)
   ...:     port.name = port.name.replace('L.', '')
   ...:     ports[port.name] = port
   ...:

In [8]: for port in ports.values():
   ...:     if port.name != 'Bodoff4':
   ...:         port.update(bs=1, log2=8, remove_fuzz=True, padding=1)
   ...:     else:
   ...:         port.update(bs=1/8, log2=16, remove_fuzz=True, padding=2)
   ...:     port.density_df = port.density_df.apply(lambda x: np.round(x, 14))
   ...:     qd(port)
   ...:     print(port.name)
   ...:     print('='*80 + '\n')
   ...:

             E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
unit   X
wind1  Freq     1                           0
       Sev   19.8     19.8 -2.2204e-16      2         2     1.5         1.5
       Agg   19.8     19.8 -2.2204e-16      2         2     1.5         1.5
quake1 Freq     1                           0
       Sev      5        5  8.8818e-16 4.3589    4.3589  4.1295      4.1295
       Agg      5        5  8.8818e-16 4.3589    4.3589  4.1295      4.1295
total  Freq     2                           0
       Sev   12.4     12.4           0 2.6458            2.2679
       Agg   24.8     24.8           0 1.8226    1.8226  1.4715      1.4715
log2 = 8, bandwidth = 1, validation: not unreasonable.
Bodoff1
================================================================================


             E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
unit   X
wind2  Freq     1                           0
       Sev     10       10 -2.2204e-16      2         2     1.5         1.5
       Agg     10       10 -2.2204e-16      2         2     1.5         1.5
quake2 Freq     1                           0
       Sev      5        5  8.8818e-16 4.3589    4.3589  4.1295      4.1295
       Agg      5        5  8.8818e-16 4.3589    4.3589  4.1295      4.1295
total  Freq     2                           0
       Sev    7.5      7.5  2.2204e-16 2.8087            2.8984
       Agg     15       15  2.2204e-16  1.972     1.972  2.1153      2.1153
log2 = 8, bandwidth = 1, validation: not unreasonable.
Bodoff2
================================================================================


             E[X] Est E[X]    Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
```

(continues on next page)

(continued from previous page)

```
unit   X
wind3  Freq     1                           0
       Sev      1        1 -2.2204e-16      2         2     1.5         1.5
       Agg      1        1 -2.2204e-16      2         2     1.5         1.5
quake3 Freq     1                           0
       Sev      5        5  8.8818e-16 4.3589    4.3589  4.1295      4.1295
       Agg      5        5  8.8818e-16 4.3589    4.3589  4.1295      4.1295
total  Freq     2                           0
       Sev      3        3  6.6613e-16 5.2015            5.9989
       Agg      6        6  8.8818e-16 3.6477    3.6477   4.079       4.079
log2 = 8, bandwidth = 1, validation: not unreasonable.
Bodoff3
================================================================================


            E[X]  Est E[X]     Err E[X]   CV(X)  Est CV(X)   Skew(X)  Est Skew(X)
unit  X
a     Freq  0.25                           2                  2
      Sev      4   3.9998 -4.0689e-05      1    1.0001        2       1.9995
      Agg      1  0.99996 -4.0689e-05 2.8284    2.8286   4.2426       4.2426
b     Freq  0.05                      4.4721             4.4721
      Sev     20       20 -1.6276e-06      1         1        2            2
      Agg      1        1 -1.6276e-06 6.3246    6.3246   9.4868       9.4868
c     Freq  0.05                      4.4721             4.4721
      Sev    100      100 -6.5104e-08      1         1        2            2
      Agg      5        5 -6.5104e-08 6.3246    6.3246   9.4868       9.4868
total Freq  0.35                      1.6903             1.6903
      Sev     20       20 -6.0917e-06 2.5467             5.3022
      Agg      7        7 -6.0918e-06 4.6247    4.6247   8.9162       8.9162
log2 = 16, bandwidth = 1/8, validation: not unreasonable.
Bodoff4
================================================================================
```

5.6.7 Thought Experiment Number 1

There are four possible events ω, leading to the loss outcomes $X(\omega)$ laid out next.

Event, ω	X_1	X_2	X	$\Pr(\omega)$	F	S
No loss	0	0	0	0.76	0.76	0.24
Wind	99	0	99	0.19	0.95	0.05
Quake	0	100	100	0.04	0.99	0.01
Both	99	100	199	0.01	1.00	0.00

Compute the allocation using all the methods. In the next block, `EX` shows expected unlimited loss by unit. `sa VaR` and `sa TVaR` show stand-alone 0.99 VaR and TVaR. The remaining rows display results for the methods just described. The apparent issue with the `coTVaR` method is caused by the probability mass at 100. A `co ES` allocation would re-scale the `coTVaR` allocation shown.

```
In [9]: port = ports['Bodoff1']

In [10]: reg_p = 0.99

In [11]: a = port.q(reg_p, 'lower')

In [12]: print(f'VaR assets = {a}')
VaR assets = 100.0

In [13]: basic = bodoff_exhibit(port, reg_p)
```

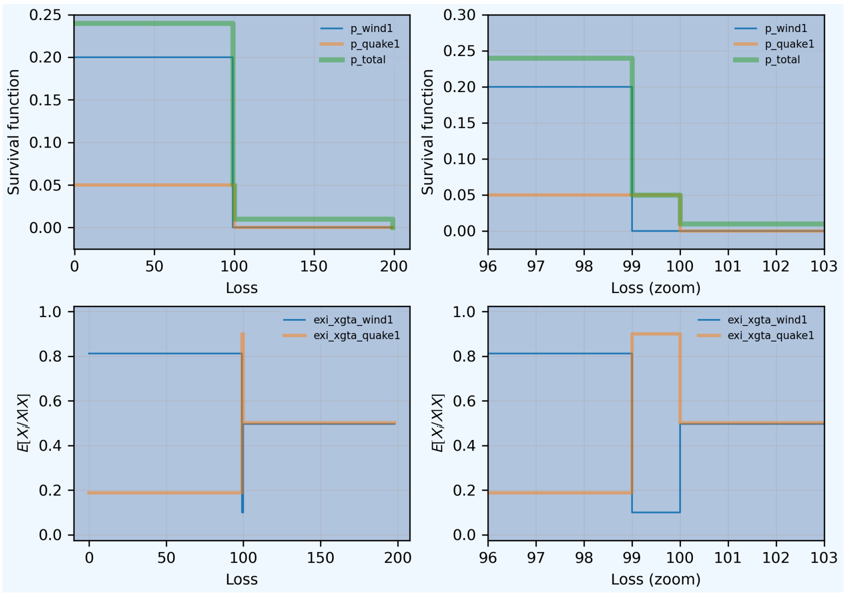
(continues on next page)

(continued from previous page)

```
In [14]: qd(basic, col_space=10)

                    wind1      quake1       total
method
EX                      5        24.8        19.8
sa VaR                 99         100         100
sa TVaR                99         100         199
pct EX             25.253      125.25         100
coVaR                   0         100         100
alt coVaR          9.9497       90.05         100
naive coTVaR       16.528      83.472         100
coTVaR               82.5      20.833      103.33
plc                80.527      19.473         100
```

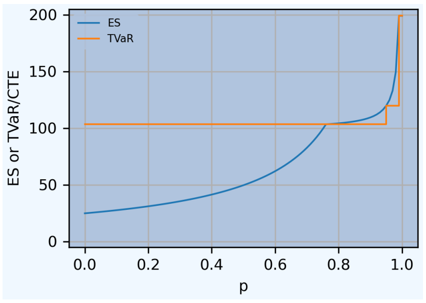
Graphs of the survival and allocation functions for Bodoff Example 1. Top row: survival functions, bottom row: $\alpha_i(x)$ allocation functions. Left side shows full range of $0 \le x \le 200$ and right side highlights the functions around the loss points, $96 \le x \le 103$.

```
In [15]: fig, axs = plt.subplots(2, 2, figsize=(2 * 3.5, 2 * 2.45), constrained_
↪layout=True)

In [16]: ax0, ax1, ax2, ax3 = axs.flat

In [17]: df = port.density_df

In [18]: for ax in axs.flat[:2]:
   ....:     (1 - df.query('(S>0 or p_total>0) and loss<=210').filter(regex='p_').
↪cumsum()).\
   ....:         plot(drawstyle="steps-post", ax=ax, lw=1)
   ....:     ax.lines[1].set(lw=2, alpha=.5)
   ....:     ax.lines[2].set(lw=3, alpha=.5)
   ....:     ax.grid(lw=.25)
   ....:     ax.legend(loc='upper right')
   ....:

In [19]: ax0.set(ylim=(-0.025, .25), xlim=(-.5, 210), xlabel='Loss', ylabel=
↪'Survival function');

In [20]: ax1.set(ylim=(-0.025, .3), xlim=[96,103], xlabel='Loss (zoom)', ylabel=
↪'Survival function');

In [21]: for ax in axs.flat[2:]:
   ....:     df.query('(S>0) and loss<=210').filter(regex='exi_xgta_[wq]').
↪plot(drawstyle="steps-post", lw=1, ax=ax)
   ....:     ax.lines[1].set(lw=2, alpha=.5)
   ....:     ax.grid(lw=.25)
   ....:     ax.legend(loc='upper right')
   ....:

In [22]: ax2.set(ylim=(-0.025, 1.025), xlabel='Loss', ylabel='$E[X_i/X | X]$');

In [23]: ax3.set(ylim=(-0.025, 1.025), xlim=(96,103), xlabel='Loss (zoom)', ylabel=
↪'$E[X_i/X | X]$');
```



Expected Shortfall (usually called TVaR) differs from Bodoff's Tail Value at Risk (generally called CTE) for a discrete distribution. TVaR/CTE is a jump function. ES is a continuous, increasing function taking all values between the mean and maximum value of X. The graph illustrates the functions for Bodoff Example 1.

```
In [24]: fig, ax = plt.subplots(1, 1, figsize=(3.5, 2.45), constrained_layout=True)

In [25]: ps = np.linspace(0, 1, 101)

In [26]: tp = port.tvar(ps)

In [27]: ax.plot(ps, tp, lw=1, label='ES');

In [28]: ax.plot(df.F, port.density_df.exgta_total, lw=1, label='TVaR', drawstyle=
↪'steps-post');

In [29]: ax.plot([0, .76], [port.ex/.24, port.ex/.24, ], c='C1', lw=1, label=None);

In [30]: ax.grid();

In [31]: ax.legend();

In [32]: ax.set(ylim=[-5, 205], xlabel='p', ylabel='ES or TVaR/CTE');
```



5.6.8 Bodoff Examples 1-3

Example 2 illustrates that `plc` can produce an answer that is different from expected losses. Example 3 it illustrates fungibility of pooled capital, with losses from X_1 covered by the total premium. `coTVaR` suffers the same issues in Examples 2 and 3 as it does in Example 1.

```
In [33]: basic1 = bodoff_exhibit(ports['Bodoff1'], reg_p)

In [34]: basic2 = bodoff_exhibit(ports['Bodoff2'], reg_p)

In [35]: basic3 = bodoff_exhibit(ports['Bodoff3'], reg_p)

In [36]: basic_all = pd.concat((basic1, basic2, basic3), axis=1,
   ....:     keys=[f'Ex {i}' for i in range(1,4)])
   ....:

In [37]: qd(basic_all, col_space=7)

                 Ex 1                        Ex 2                        Ex 3           ␣
↪
               wind1  quake1   total   wind2  quake2   total   wind3  quake3  ␣
↪total
method                                                                          ␣
↪
EX                 5    24.8    19.8       5      15      10       5       6    ␣
↪1
sa VaR            99     100     100      50     100     100       5     100  ␣
↪100
sa TVaR           99     100     199      50     100     150       5     100  ␣
↪105
pct EX        25.253  125.25     100      50     150     100     500     600  ␣
↪100
coVaR              0     100     100      -0     100     100       0     100  ␣
↪100
alt coVaR     9.9497   90.05     100  6.6667  93.333     100 0.95238  99.048  ␣
↪100
naive coTVaR  16.528  83.472     100  9.0909  90.909     100  0.9901   99.01  ␣
↪100
coTVaR          82.5  20.833  103.33      10     100     110       1     100  ␣
↪101
plc           80.527  19.473     100  43.611  56.389     100   4.873  95.127  ␣
↪100
```

5.6.9 Bodoff Example 4

The next table recreates the exhibit in Section 9.1 of Bodoff's paper. There are three units labelled `a`, `b`, and `c`. It shows the percent allocation of capital to each unit across different methods. Breakeven percentile equals the percentile equal to expected losses. Bodoff's calculation uses 10,000 simulations. The table shown here uses FFTs to obtain a close-to exact answer. The exponential distribution is borderline thick tailed, and so is quite hard to work with for both simulation methods and FFT methods.

```
In [38]: p4 = ports['Bodoff4']

In [39]: df91 = pd.DataFrame(columns=list('abc'), dtype=float)

In [40]: tv = p4.var_dict(.99, 'tvar')

In [41]: df91.loc['sa TVaR 0.99'] = np.array(list(tv.values())[:-1]) / sum(list(tv.
↪values())[:-1])
```

(continues on next page)

(continued from previous page)

```
In [42]: pbe = float(p4.cdf(p4.ex))

In [43]: for p in [.99, .95, .9, pbe]:
   ....:     tv = p4.cotvar(p)
   ....:     df91.loc[f'naive TVaR {p:.3g}'] = tv[:-1] / tv[-1]
   ....:

In [44]: v = ((p4.density_df.filter(regex='exi_xgta_[abc]').
   ....:                     shift(1).cumsum() * p4.bs).loc[p4.q(.99)]).values
   ....:

In [45]: df91.loc['plc'] = v / v.sum()

In [46]: df91.index.name = 'line'

In [47]: qd(df91, col_space=10, float_format=lambda x: f'{x:.1%}')

                           a          b          c
line
sa TVaR 0.99            5.5%      15.8%      78.8%
naive TVaR 0.99         0.4%       0.7%      98.9%
naive TVaR 0.95         1.3%      11.3%      87.5%
naive TVaR 0.9          6.7%      14.7%      78.6%
naive TVaR 0.876        9.0%      14.7%      76.3%
plc                     4.5%       9.6%      85.8%
```

Pricing for Bodoff Example 4

Bodoff Example 4 is based on a three unit portfolio. Each unit has a Bernoulli 0/1 frequency and exponential severity:

- Unit `a` has a 0.25 probability of a claim and 4 severity
- Unit `b` has a 0.05 probability of a claim and 20 severity
- Unit `c` has a 0.01 probability of a claim and 100 severity

All units have unlimited expectation 1.0

Bodoff does not consider pricing per se. His allocation can be considered as $P_i + Q_i$, with no opinion on the split between margin and equity. Making additional assumptions we can compare the `plc` capital allocation with other methods. Assume total roe = 0.1 at 0.99-VaR capital standard. Set up the target return, premium, and regulatory capital threshold (99% VaR):

```
In [48]: roe = 0.1

In [49]: reg_p = 0.99

In [50]: v = 1 / (1 + roe)

In [51]: d = 1 - v

In [52]: port = ports['Bodoff4']

In [53]: a = port.q(reg_p)

In [54]: el = port.density_df.at[a, 'lev_total']

In [55]: premium = v * el + d * a

In [56]: q = a - premium
```

(continues on next page)

(continued from previous page)

```
In [57]: margin = premium - el

In [58]: roe, a, el, port.ex, premium, el / premium, q, margin / q
Out[58]:
(0.1,
 164.875,
 5.97612830432361,
 6.999957357424143,
 20.42148027665783,
 0.29263932992920433,
 144.45351972334217,
 0.10000000000000003)
```

Calibrate pricing distortions to required return.

```
In [59]: port.calibrate_distortions(ROEs=[roe], Ps=[reg_p], strict='ordered');

In [60]: qd(port.distortion_df)

                                 S      L      P      PQ      Q  COC   param      ␣
↪error
a        LR       method                                                          ␣
↪
164.875 292.639m ccoc    0.0099961 5.9761 20.421 0.14137 144.45  0.1      0.1      ␣
↪   0
                  ph     0.0099961 5.9761 20.421 0.14137 144.45  0.1 0.60427  1.
↪7693e-10
                  wang   0.0099961 5.9761 20.421 0.14137 144.45  0.1 0.69321  3.
↪3953e-06
                  dual   0.0099961 5.9761 20.421 0.14137 144.45  0.1  3.8032 -2.
↪7904e-08
                  tvar   0.0099961 5.9761 20.421 0.14137 144.45  0.1 0.70736  4.
↪3553e-06
```

Allocate premium plus equity to each unit across different pricing methods. All methods except percentile layer capital calibrated to the same total premium and capital level. Distortions that price tail loss will allocate the most to unit `c`, the most volatile. More bowed distortions will allocate most to `a`. The three units have the same expected loss (last row). `covar` is covariance method; `coVaR` is conditional VaR. `agg` corresponds to the PIR approach and `bod` to Bodoff's methods. Only additive methods are shown. `method` ordered by allocation to unit `a` the least skewed; `c` is the most skewed.

```
In [61]: ad_ans = port.analyze_distortions(p=reg_p, kind='lower')

In [62]: basic = bodoff_exhibit(port, reg_p)

In [63]: qd(basic, col_space=10)

                      a          b          c      total
method
EX                    1          1          5          7
sa VaR               14       32.5      162.5     164.88
sa TVaR           18.42     52.989     264.94     267.26
pct EX           23.554     23.554     117.77     164.88
coVaR            1.0864     2.7956     160.99     164.88
alt coVaR       0.74288     1.3108     162.82     164.87
naive coTVaR    0.66867     1.1279     163.08     164.88
coTVaR           1.1127     7.0366     156.86     165.01
plc              7.4527     15.899     141.52     164.87

In [64]: ans = pd.concat((ad_ans.comp_df.xs('P', 0, 1) + ad_ans.comp_df.xs('Q', 0,␣
↪1),
```

(continues on next page)

(continued from previous page)

```
   ....:                  basic.rename(columns=dict(X='total')).iloc[3:]), keys=(
↪'agg', 'bod'))
   ....:

In [65]: if port.name[-1] in list('123'):
   ....:     ans = ans.sort_values('X1')
   ....:     bit = ans.query(' abs(total - @a) < 1e-3 and abs(X1 + X2 - total) <␣
↪1e-3 ').dropna()
   ....:

In [66]: if port.name[-1] not in list('123'):
   ....:     ans = ans.sort_values('a')
   ....:     bit = ans.query(' abs(total - @a) < 1e-2 and abs(a + b + c - total) <␣
↪1e-2 ')
   ....:

In [67]: bit.index.names =['approach', 'method']

In [68]: qd(bit, col_space=10)

                                 a          b          c      total
approach   method
bod        naive coTVaR    0.66867     1.1279     163.08     164.88
           alt coVaR       0.74288     1.3108     162.82     164.87
           coVaR            1.0864     2.7956     160.99     164.88
           plc              7.4527     15.899     141.52     164.87
           pct EX           23.554     23.554     117.77     164.88
```

Premium for PIR and Bodoff methods, sorted by premium for a. All methods produce the same total premium by calibration. Very considerable differences are evident across the methods.

```
In [69]: basic.loc['EXa'] = \
   ....: port.density_df.filter(regex='exa_[abct]').loc[a].rename(index=lambda x:␣
↪x.replace('exa_', ''))
   ....:

In [70]: premium_df = basic.drop(index=['EX', 'sa TVaR', 'coTVaR'])

In [71]: premium_df = premium_df.loc['EXa'] * v + d * premium_df

In [72]: ans = pd.concat((ad_ans.comp_df.xs('P', 0, 1), premium_df),
   ....:    keys=('agg', 'bod')).sort_values('a')
   ....:

In [73]: bit = ans.query(' abs(total - @premium) < 1e-2 and abs(a + b + c - total)
↪< 1e-2 ')

In [74]: bit.index.names =['approach', 'method']

In [75]: qd(bit, col_space=10, sparsify=False)

                                 a          b          c      total
approach   method
bod        naive coTVaR    0.96674     1.0069     18.448     20.421
bod        alt coVaR       0.97349     1.0236     18.424     20.421
bod        coVaR            1.0047     1.1585     18.258     20.421
agg        Dist ph          1.5177     2.1557     16.756     20.429
bod        plc              1.5835     2.3497     16.488     20.421
agg        Dist wang        1.8859     2.5978     15.944     20.428
agg        Dist dual        2.6728     3.2827     14.471     20.426
bod        pct EX           3.0472     3.0456     14.329     20.421
```

(continues on next page)

(continued from previous page)

```
agg         Dist tvar          3.4054      3.3995      13.621      20.426
```

Corresponding loss ratios (remember, these are cat lines).

```
In [76]: bit_lr = premium_df.loc['EXa'] / bit

In [77]: qd(bit_lr, col_space=10, sparsify=False,
   ....:    float_format=lambda x: f'{x:.1%}')
   ....:

                                  a           b           c       total
approach    method
bod         naive coTVaR     103.1%       98.8%       21.6%       29.3%
bod         alt coVaR        102.4%       97.2%       21.6%       29.3%
bod         coVaR             99.2%       85.9%       21.8%       29.3%
agg         Dist ph           65.7%       46.1%       23.8%       29.3%
bod         plc               62.9%       42.3%       24.2%       29.3%
agg         Dist wang         52.8%       38.3%       25.0%       29.3%
agg         Dist dual         37.3%       30.3%       27.5%       29.3%
bod         pct EX            32.7%       32.7%       27.8%       29.3%
agg         Dist tvar         29.3%       29.3%       29.3%       29.3%
```

5.6.10 Bodoff Summary

Bodoff's methods allocate all capital like loss and do not distinguish expected loss, margin and equity. It does not get to a price. It is event-centric, allocating to **events**, but really allocating to **peril=lines**. Premium is not mentioned until Section 7 (of 10). Then, it uses the basic CCoC formula $P = vL + da$ (eq. 8.2).

5.6.11 CAS Exam Question: Spring 2018 Question 15

An insurer has exposure to two independent perils, wind and earthquake:

- Wind has a 15% chance of a \$5 million loss, and an 85% chance of no loss.
- Earthquake has a 1 % chance of a \$15 million loss, and a 99% chance of no loss.

Using the capital allocation by percentile layer methodology with a 99.5% VaR capital requirement, determine how much capital should be allocated to each peril.

Solution.

The last row gives the percentile layer capital.

```
In [78]: cas15 = build('port CASq15 '
   ....:     'agg X1 1 claim dsev [0,  5] [0.85, 0.15] fixed '
   ....:     'agg X2 1 claim dsev [0, 15] [0.99, 0.01] fixed ')
   ....:

In [79]: qd(cas15)

            E[X] Est E[X]   Err E[X]  CV(X) Est CV(X) Skew(X) Est Skew(X)
unit  X
X1    Freq     1                          0
      Sev   0.75      0.75 2.2204e-16 2.3805    2.3805  1.9604      1.9604
      Agg   0.75      0.75 2.2204e-16 2.3805    2.3805  1.9604      1.9604
X2    Freq     1                          0
      Sev   0.15      0.15 8.8818e-16 9.9499    9.9499  9.8494      9.8494
      Agg   0.15      0.15 8.8818e-16 9.9499    9.9499  9.8494      9.8494
total Freq     2                          0
```

(continues on next page)

(continued from previous page)

```
      Sev    0.45      0.45 2.2204e-16 3.7168            4.7835
      Agg     0.9       0.9 2.2204e-16 2.5856    2.5856  3.4839      3.4839
log2 = 16, bandwidth = 1/128, validation: not unreasonable.

# cas15.update(bs=1, log2=8, remove_fuzz=True, padding=1)
In [80]: cas15.density_df = cas15.density_df.apply(lambda x: np.round(x, 10))

In [81]: basic = bodoff_exhibit(cas15, reg_p=.995)

In [82]: qd(basic, col_space=10)

                     X1          X2       total
method
EX                 0.75        0.15         0.9
sa VaR                5          15          15
sa TVaR               5          15        16.5
pct EX             12.5         2.5          15
coVaR                 0          15          15
alt coVaR        0.5625      14.438          15
naive coTVaR    0.71429      14.286          15
coTVaR             0.75          15       15.75
plc              5.0714      9.9286          15

In [83]: df = cas15.density_df.query('S > 0 or p_total > 0')
```

The calculation of `plc` as the integral of α for unit 1 is simply:

```
In [84]: df.exi_xgta_X1.shift(1, fill_value=0).cumsum().loc[15] * cas15.bs
Out[84]: 5.071372239500204
```

5.7 The Pollaczeck-Khinchine Formula

Objectives: Illustrate and compute Pollaczeck-Khinchine formula for probability of eventual ruin for a compound Poisson process.

Audience: Advanced users.

Prerequisites: Advanced risk theory.

Contents:

5.7.1 Helpful References

- Embrechts *et al.* [1997] Section 1.2
- Mildenhall and Major [2022] sections 8.4.2 and 9.3 and references therein (largely reproduced here).

5.7.2 Classical Risk Theory and the Pollaczeck-Khinchine Formula

The Pollaczeck-Khinchine formula determines the probability of eventual ruin in a portfolio where claims are driven by a compound Poisson process, in terms of starting surplus and the premium rate. Losses are generated by a Poisson process with λ annual expected claims and iid severity X. Losses up to time t are given by

$$A(t) = X_1 + \cdots + X_{N(t)},$$

where $N(t)$ is Poisson with mean λt. Expected loss per year equals $\lambda\mathsf{E}[X]$. Premium per year equals $(1+r)\lambda\mathsf{E}[X]$ where r is the ratio of profit to expected loss. The corresponding expected loss ratio is $1/(1+r)$. If $r \le 0$ then eventual ruin is certain, so assume $r > 0$.

Define the **integrated severity distribution** by

$$\begin{aligned} F_I(x) &= \frac{1}{\mathsf{E}[X]}\int_0^x S(t)dt \\ &= \frac{\mathsf{LEV}(x)}{\mathsf{E}[X]} \\ &= 1 - \frac{\mathsf{E}[(X-x)^+]}{\mathsf{E}[X]} \end{aligned}$$

where S is the survival function of X. F_I is a thicker tailed distribution than F. Let

$$U_{u,r}(t) = u + (1+r)\lambda\mathsf{E}[X]t - A(t)$$

denote accumulated surplus to time t given starting surplus u. U is called the **surplus process**. Finally, let

$$\psi(u,r) = \Pr(U_{u,r}(t) < 0 \text{ for some } t \ge 0)$$

be the probability of eventual ruin.

The **Pollaczeck-Khinchine formula** says that

$$\psi(u,r) = 1 - \frac{r}{1+r}\sum_{n\ge 0}(1+r)^{-n}F_I^{n*}(u)$$

where F_I^{n*} is distribution of the sum of n independent variables with distribution F_I. Note that

$$G(z) = \frac{r}{1+r}\sum_{n\ge 0}\frac{z^n}{(1+r)^n} = \frac{r}{1+r-z}$$

is the probability generating function of a geometric distribution M with mean $1/r$ and $\Pr(M=m) = \frac{r}{1+r}\frac{1}{(1+r)^m}$. Therefore $\psi(u) = \Pr(Y > u)$ where Y is an aggregate distribution with frequency M and severity F_I. A surprising consequence is that the probability of eventual ruin starting with no surplus, $\psi(0) = 1 - \Pr(Y=0) = 1 - \Pr(M=0) = \frac{1}{1+r}$ equals the expected loss ratio!

Embrechts *et al.* [1997] Section 1.2 shows how to derive the Pollaczeck-Khinchine formula. The key step is to determine the distribution of $X - (1+r)T$ where T is the exponential waiting time between claims, and to observe that ruin can occur only at the moment of a claim.

The Pollaczeck-Khinchine formula gives combinations of u and r that are consistent with a top-down stability requirement expressed as a target probability of eventual ruin. Overlaying a cost of capital provides a link between r and u that determines a minimum viable market size. An example of this method is given below.

Because **eventual** is the same in days, weeks or years, $\psi_{X,m}(u)$ is independent of the expected claim count λ. In unit of time $1/\lambda$ all portfolios have an expected claim count of one. Therefore $\psi^{-1}(p)$ gives a capital requirement (risk measure) that is a function of severity and not frequency, i.e., it is independent of portfolio size. Unlike most risk measures, it does not regard small portfolios as more risky than large ones.

The **Cramer-Lundberg formula** is an approximation to ψ that applies for thin tailed severities. It says that

$$\psi(u,r) \le e^{-ku}$$

where $k>0$ is a constant called the **adjustment coefficient** solving

$$e^{kP} = \mathsf{E}[e^{kA(1)}]$$

where $P=(1+r)\lambda\mathsf{E}[X]$ is the premium. Given a top-down stability requirement, we can work backwards from the Cramer-Lundberg formula to determine a premium.

Exercise. Show that if $k=-\log(p)/u$ and premium

$$P = \frac{1}{k}\log\mathsf{E}[e^{kA(1)}],$$

then the Cramer-Lundberg formula ensures the probability of eventual ruin is $\le p$. The properties of P motivate the exponential premium. In turn, the approximation $P\approx \mathsf{E}[A(1)] + k\mathsf{Var}(A(1))/2$ motivates the variance principle.

Both the Cramer-Lundberg and Pollaczeck-Khinchine formulas assume independent and identically distributed severity and Poisson frequency. These can be reasonable assumptions for the loss process of a small portfolio. The case of a mixed Poisson can be decomposed as a mixture of pure Poisson processes.

5.7.3 FFT Computation

The distribution of Y can be computed using Fast Fourier transforms in the same way as any aggregate distribution. Some care is needed when the margin is very small because the claim count is very large. `Aggregate` includes `pollaczeck_khinchine()` to determine the integrated distribution F_I and convolve it with a geometric frequency.

5.7.4 Using The Pollaczeck-Khinchine Formula I

This section reproduces two examples from the risk vignette for the `actuar` package.

The first is based on a mean 10 Poisson compound with shape 2 gamma severity. The vignette uses matched-moments discretization and so our numbers do not exactly match, but they are very close. We build the compound, compute some quantiles and tvars, display the density (compare p.8-10). Then using a premium loading of 20% using the expected value premium (p.12), we reproduce the probabilities in Figure 5. Our computation is exact vs. using an approximation.

```
In [1]: from aggregate import build, qd

In [2]: import matplotlib.pyplot as plt

In [3]: a = build('agg Actuar 10 claims sev gamma 2 poisson'
   ...:           , bs=0.5, log2=8)
   ...:

In [4]: qd(a)

      E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq    10                      0.31623            0.31623
Sev      2   1.9999 -7.4963e-05 0.70711   0.71102   1.4142      1.3901
Agg     20   19.999 -7.4963e-05  0.3873   0.38801   0.5164     0.51634
log2 = 8, bandwidth = 1/2, validation: fails sev cv, agg cv.
```

(continues on next page)

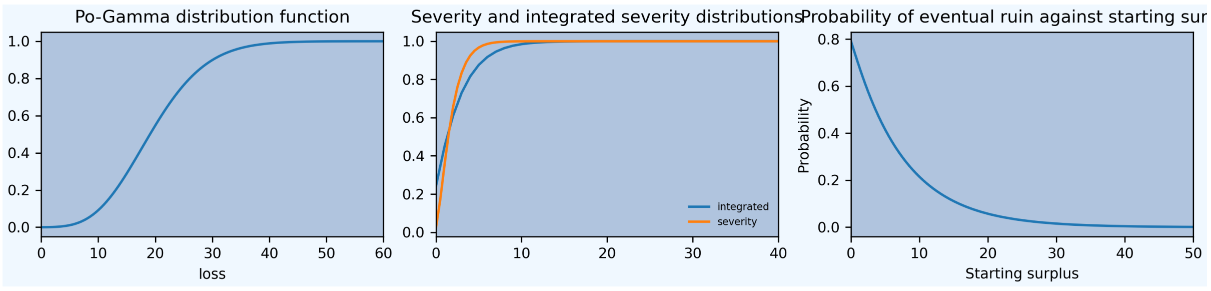
(continued from previous page)

```
In [5]: ps = [0.25, .5, .75, .9, .95, .975, .99, .995, .999, 1-1e-14]

In [6]: qd(pd.DataFrame({'p': ps, 'q': a.q(ps)}), float_format=lambda x: f'{x:10.
↪3f}' if x < 1 else f'{x:10.1f}')

           p          q
0      0.250       14.5
1      0.500       19.5
2      0.750       25.0
3      0.900       30.5
4      0.950       34.0
5      0.975       37.0
6      0.990       41.0
7      0.995       43.5
8      0.999       49.5
9      1.000      115.5

In [7]: fig, axs = plt.subplots(1, 3, figsize=(3 * 3.5, 2.45), constrained_
↪layout=True)

In [8]: ax, ax0, ax1 = axs.flat

In [9]: a.density_df.F.plot(ax=ax, xlim=[0, 60], title='Po-Gamma distribution␣
↪function');

In [10]: qd(a.density_df.p_total.head(20).reset_index(drop=False), float_
↪format=lambda x: f'    {x:<12.5g}' if 0 < x < .1 else f'{x:10.1f}')

         loss         p_total
0         0.0    5.9175e-05
1         0.5    8.6904e-05
2         1.0    0.00017152
3         1.5    0.00028809
4         2.0    0.00045063
5         2.5    0.00066863
6         3.0    0.0009508
7         3.5    0.0013052
8         4.0    0.001739
9         4.5    0.0022578
10        5.0    0.0028658
11        5.5    0.0035653
12        6.0    0.0043564
13        6.5    0.0052372
14        7.0    0.0062034
15        7.5    0.0072487
16        8.0    0.0083649
17        8.5    0.0095419
18        9.0    0.010768
19        9.5    0.01203

In [11]: qd(a.tvar([.9, .95, .99]))
[35.02023434 38.14387968 44.6199146 ]

In [12]: ruins, find_us, mean, dfi  = a.cramer_lundberg(.2)

In [13]: ax0.plot(np.cumsum(dfi), label='integrated')
Out[13]: [<matplotlib.lines.Line2D at 0x7f8089819ed0>]

In [14]: ax0.plot(a.density_df.p_sev.cumsum(), label='severity')
Out[14]: [<matplotlib.lines.Line2D at 0x7f8089819b10>]
```

(continues on next page)

(continued from previous page)

```
In [15]: ax0.set(xlim=[0, 40], title='Severity and integrated severity␣
↪distributions')
Out[15]: [(0.0, 40.0), Text(0.5, 1.0, 'Severity and integrated severity␣
↪distributions')]

In [16]: ax0.legend(loc='lower right')
Out[16]: <matplotlib.legend.Legend at 0x7f8087ca3a90>

In [17]: ruins.plot(ax=ax1, xlim=[0, 50],
   ....:            title='Probability of eventual ruin against starting surplus',
   ....:            ylabel='Probability', xlabel='Starting surplus');
   ....:
```



The second uses a Pareto severity, where the integrated distribution can be computed exactly. The following code produces the exact values for the probability of eventual default against starting surplus. All the values fall in the range between lower and upper shown on p.19.

```
In [18]: a = build('agg Actuar2 1 claim sev 4 * pareto 5 - 4 fixed')

In [19]: qd(a)

      E[X] Est E[X]    Err E[X]  CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq     1                           0
Sev      1        1 -4.2872e-06  1.291    1.2907   4.6476      4.5872
Agg      1        1 -4.2872e-06  1.291    1.2907   4.6476      4.5872
log2 = 16, bandwidth = 1/512, validation: fails sev skew, agg skew.

In [20]: ruins, find_us, mean, dfi  = a.cramer_lundberg(.2)

In [21]: ruins.name = 'Prob'

In [22]: bit = ruins.loc[np.arange(0, 51, 5)].to_frame()

In [23]: bit.index = bit.index.astype(int)

In [24]: bit.index.name = 'Initial surplus'

In [25]: qd(bit, ff=lambda x: f'{x:.5f}')

                   Prob
Initial surplus
0               0.83306
5               0.42670
10              0.23477
15              0.13149
20              0.07439
25              0.04242
30              0.02435
35              0.01406
40              0.00818
```

(continues on next page)

(continued from previous page)

```
45              0.00479
50              0.00282
```

The actual work, to get the answer as opposed to formatting the result, is only two lines of code in Aggregate (the first and third) vs. 8 in actuar R.

5.7.5 Using The Pollaczeck-Khinchine Formula II

This section illustrates the theory using a lognormal severity with a mean of 50,000 and a CV of 10 ($\sigma = 2.15$) corresponding to a moderately risky liability line. It compares starting surplus levels for different eventual ruin probabilities assuming a margin $r = 0.1$ with a 1 million and 10 million occurrence limit. It also illustrates simulations of the surplus process in each case with starting surplus calibrated to a 0.05 probability of eventual ruin.

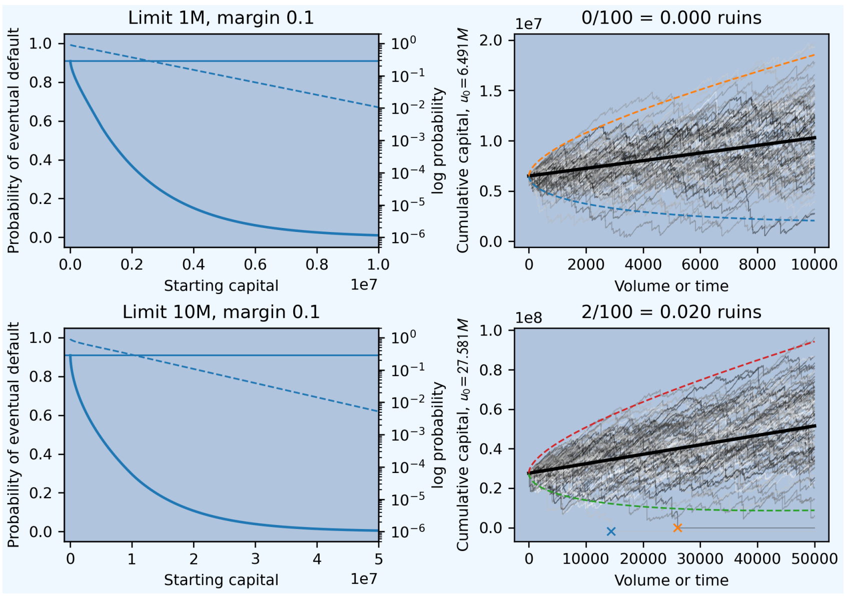
Set up the portfolio.

```
In [26]: port = build('port PZTest '
   ....:                  'agg Limit1  '
   ....:                     '0.1 claims '
   ....:                     '1000000 xs 0 '
   ....:                     'sev lognorm 50000 cv 10 '
   ....:                     'poisson'
   ....:                  'agg Limit10 '
   ....:                     '0.1 claims '
   ....:                     ' 10000000 xs 0 '
   ....:                     'sev lognorm 50000 cv 10 '
   ....:                     'poisson'
   ....:             , bs=500, log2=18, padding=1)
   ....:

In [27]: qd(port)

               E[X] Est E[X]    Err E[X]  CV(X) Est CV(X)  Skew(X) Est Skew(X)
unit    X
Limit1  Freq    0.1                      3.1623            3.1623
        Sev   38064    38060  -0.0001105 3.0931    3.0935  5.9141       5.914
        Agg  3806.4     3806  -0.0001105  10.28    10.281  18.845      18.845
Limit10 Freq    0.1                      3.1623            3.1623
        Sev   47900    47896 -8.7807e-05 5.6958    5.6963  20.969      20.969
        Agg    4790   4789.6 -8.7807e-05 18.287    18.289  64.966      64.966
total   Freq    0.2                      2.2361            2.2361
        Sev   42982    42978 -9.7853e-05 4.8898            23.502
        Agg  8596.4   8595.6 -9.7853e-05  11.16    11.161  50.728      50.728
log2 = 18, bandwidth = 500, validation: fails sev mean, agg mean.
```

The left plots show the Pollaczeck-Khinchine formula starting surplus as a function of the eventual ruin probability with margin 0.1 on linear (solid) and log (dashed) scales. The Cramer-Lundberg formula says that the probability of eventual ruin is approximately exponential, which is a straight line on a log scale.

The right column show 500 simulated surplus paths, with $\times$ indicating ruin scenarios. Capital calibrated to 0.05 eventual ruin probability. Time and volume are symmetric in the model, so volume can be regarded as time for a fixed size portfolio or a varying sized portfolio for a fixed time or a combination. Scale indicates cumulative exposure-years.

```
In [28]: from aggregate.extensions.pir_figures import fig_9_1

In [29]: fig_9_1(port)
```



The right hand plots are computed here with only 100 samples, vs. 500 used in the book, and so the approximation is not as accurate.

These simulations show that the probability of eventual ruin is constrained by the buildup of surplus in most scenarios. Defaults occur early in the simulated history. This model could be appropriate for a mutual company—indeed some mutual companies have accumulated substantial amounts of capital. For a stock company, a more realistic approach adds dividends to manage capital.

5.7.6 Market Scale and Viability

Given severity X and ratio r of margin to expected loss, the Pollaczeck-Khinchine function ψ is monotone and hence invertible, allowing us to find $u_{X,r}(p) = \psi^{-1}_{X,r}(p)$, the starting capital necessary to guarantee probability p of eventual ruin.

The amount of margin equals $r\lambda\mathsf{E}[X]$, where λ is the annual expected claim count. Since the expected margin must pay the cost of capital, we get a market viability constraint

$$r\lambda\mathsf{E}[X] \ge \iota\, u_{X,r}(p)$$

where ι is the cost of capital. Each element is influenced by different factors:

- the hazard and contract design determines X,
- the insurance product market determines r,
- the capital markets determine ι, and
- a regulator or rating agency determines (or strongly influences) p.

There are two ways to apply this formula.

First, consider a diversifying unit, such as motor liability, where insurers grow by adding new, independent insureds with the same severity. Here, the formula gives a **minimum size of market** constraint

$$\lambda \ge \iota \frac{u_{X,r}(p)}{r\mathsf{E}[X]}.$$

This function of four variables and λ is:

- Increasing and linear in ι: the market must be larger given more expensive capital.
- Decreasing in r: the market can be smaller with a higher margin.
- Decreasing in p: the market must be larger to support a stricter capital standard.

- Independent of expected severity (because ψ is homogeneous in $\mathsf{E}[X]$) but dependent on the shape of severity (which influences ψ).

It shows the natural scale using the lognormal example. Size, measured by expected annual claim count, is shown for a range of margins, limits, and stability constraints. If claim frequency is 5%, the table shows that a market with 1M limits is reasonable for all p and r. For example, the strictest stability constraint $p = 0.01$ and lowest margin rate $r = 0.025$ needs 39,814 claims, or about 800,000 policies, to be viable. With a 10M limit and same r, the market size needs to be >100,000 claims, or about 2.5 million policies, which is less achievable. However, if the margin rate increases to $r = 0.1$, the market size reduces to 9,088 claims or about 180,000 policies.

```
In [30]: from aggregate.extensions.pir_figures import natural_scale

In [31]: df = natural_scale(port)
index 262144 is out of bounds for axis 0 with size 262144

In [32]: qd(df, float_format=lambda x: f'{x:,.0f}')

Margin           25.000m   50.000m   75.000m   100.000m  150.000m  200.000m  250.
↪000m
Limit   p                                                                       ␣
↪
1.000M  10.000m     39,814    10,182     4,625     2,657     1,230       718      ␣
↪477
        50.000m     25,808     6,577     2,978     1,705       784       455      ␣
↪300
        100.000m    19,777     5,025     2,269     1,295       593       342      ␣
↪225
        250.000m    11,803     2,973     1,331       754       339       193      ␣
↪124
10.000M 10.000m        NaN    33,612    15,552     9,088     4,335     2,600     1,
↪765
        50.000m        NaN    21,571     9,916     5,758     2,714     1,610     1,
↪079
        100.000m       NaN    16,386     7,489     4,324     2,015     1,182      ␣
↪794
        250.000m       NaN     9,531     4,280     2,427     1,101       614      ␣
↪382
```

Note: these numbers differ slightly from the book because of the update parameters used for `port`.

Second, consider a non-diversifying unit, writing catastrophe exposed business, where insurers grow by covering a greater proportion of each event. Severity becomes market share times an industry severity X, and the number of events is fixed. US hurricane reinsurance is an example. In this case, viability is independent of market share and is controlled by whether the inequality has a solution that is acceptable to both the product market and the capital market. Viability is harder to achieve

- with smaller λ: rare events are more difficult to insure,
- with lower p: higher quality insurance is more expensive,
- with higher ι: more costly capital, and
- with lower r because u increases quickly.

5.8 Calculations For Each `aggregate` Class

Objectives: Describe calculations performed by the `Aggregate`, `Portfolio`, `Distortion`, and `Bounds` classes.

Audience: Advanced users and programmers.

Prerequisites: DecL, general use of `aggregate`, probability.

See also: *API Reference*, *A Ten Minute Guide to aggregate*.

Contents:

5.8.1 Helpful References

- Mildenhall and Major [2022]
- Wang [1995]
- Wang [1996]
- Mildenhall [2022]

5.8.2 `Aggregate` Class Calculations

Todo: Discussion to follow.

5.8.3 `Portfolio` Class Calculations

A `Portfolio` is a collection of `Aggregate` objects. The class computes the densities of each aggregate component as well as the sum, and also computes the variables shown below. These variables are central to many allocation algorithms. All computations use FFTs to compute relevant convolutions and surface integrals. $X_i(a$ represents recoveries to line i when total capital is a and lines have equal priority. It is given by $X_i(a) = X_i(X \wedge a)/X$: when $X \le a$ line i is paid in full and $X_i(a) = X_i$ and when $X > a$ payments are a pro rata X_i/X share of available assets a. Hence expected recoveries are

$$
\begin{aligned}
\mathsf{E}[X_i(a)] &= \mathsf{E}[X_i(X \wedge a)/X] \\
&= \mathsf{E}[X_i(X \wedge a)/X \mid X \le a]F(a) + \mathsf{E}[X_i(X \wedge a)/X \mid X > a]S(a) \\
&= \mathsf{E}[X_i \mid X \le a]F(a) + a\mathsf{E}[X_i/X \mid X > a]S(a) \\
&= \mathsf{E}[X_i \mid X \le a]F(a) + a\alpha_i(a)S(a)
\end{aligned}
$$

emphasizing the importance of knowing $\mathsf{E}[X_i/X \mid X]$.

Densities are computed using FFT in $O(n \log(n))$ time.

Table 5: Variables and computational complexity by line i

Variable	Meaning	Computation	Complexity
All lines combined			
p_total	Density of $X = \sum_i X_i$	FFT Convolution	
exa_total	$\mathsf{E}[\min(X, a)] = \mathsf{E}[X \wedge a]$	Cumsum of S	$O(n)$
exlea_total	$\mathsf{E}[X \mid X \le a]$		
exgta_total	$\mathsf{E}[X \mid X > a]$		
By line			
p_line	Density of X_i	FFT computation of aggregate using MGF	
exeqa_line	$\mathsf{E}[X_i \mid X = a]$	Conv $xf_i(x)$, $f_{\hat{i}}$	$O(n\log(n))$
lev_line	$\mathsf{E}[\min(X_i, a)] = \mathsf{E}[X_i \wedge a]$	Cumsum of S_i	$O(n)$
e2pri_line	$\mathsf{E}[X_{i,2}(a)]$	Conv $\mathsf{E}[X_i \wedge x]$, $f_{\hat{i}}$	$O(n\log(n))$
exlea_line	$\mathsf{E}[X_i \mid X \le a]$	Cumsum of $E(X_i \mid X = x)f_X(x)$	$O(n)$
e_line	$\mathsf{E}[X_i]$		
exgta_line	$\mathsf{E}[X_i \mid X > a]$	Conditional expectation formula	
exi_x_line	$\mathsf{E}[X_i/X]$	Sum using conditional expectation	
exi_xlea_line	$\mathsf{E}[X_i/X \mid X \le a]$	Cumsum of $\mathsf{E}[X_i \mid X = x]f_X(x)/x$	
exi_xgta_line	$\mathsf{E}[X_i/X \mid X > a]$	Conditional expectation formula	
exa_line	$\mathsf{E}[X_i(a)]$	Conditional expectation formula	
epd_i_line	$(\mathsf{E}[X_i] - \mathsf{E}[X \wedge a))/\mathsf{E}[X_i]$	Stand-alone Expected Policyholder Deficit	
epd_i_line	$(\mathsf{E}[X_i] - \mathsf{E}[X_i(a)]/\mathsf{E}[X_i]$	Equal priority EPD	
epd_i_line	$(\mathsf{E}[X_i] - \mathsf{E}[X_{i,2}(a)]/\mathsf{E}[X_i]$	Second priority EPD	

For Total, All Lines X

- Density f computed by convolving each individual line using FFTs.
- F and S are computed from the cumulative sums of the density.
- exa_total $= \mathsf{E}[\min(X, a)] = \mathsf{E}[X \wedge a]$, also called lev_total for limited expected value, is computed as cumulative sums of S times bucket size. Note exa_total= lev_total.
- exlea_total $= \mathsf{E}[X \mid X \le a]$ is computed using the relation $\mathsf{E}[X \wedge a] = \int_0^a tf(t)dt + aS(a)$ as

$$\mathsf{E}[X \mid X \le a] = \frac{1}{F(a)}\int_0^a tf(t)dt = \frac{\mathsf{E}[X \wedge a] - aS(a)}{F(a)}.$$

  When $F(a)$ is very small these values are unreliable and so the first values are set equal to zero.

- exgta_total $= \mathsf{E}[X \mid X > a]$ is computed using the relation $\mathsf{E}[X] = \mathsf{E}[X \mid X \le a]F(a) + \mathsf{E}[X \mid X > a]S(a)$. Therefore

$$\mathsf{E}[X \mid X > a] = \frac{\mathsf{E}[X] - \mathsf{E}[X \mid X \le a]F(a)}{/S(a)}.$$

For Individual Lines X_i

- Density and distributions as for total.
- exeqa_line $= \mathsf{E}[X_i \mid X = a] = \kappa_i(a)$ can be computed efficiently using FFTs in the case X_i are independent. Without loss of generality $X = X_i + \hat{X}_i$ where $\hat{X}_i$ is the sum of all other lines ("not i"). Let $f_x(x_i, \hat{x}_i)$ be the conditional density of $X_i = x_i$, $\hat{X}_i = \hat{x}_i$ given $X = x$. Thus $f_x(x_i, \hat{x}_i) = f(x_i, \hat{x}_i)/f_X(x)$ where f is the bivariate density of X_i and $\hat{X}_i$ and f_X is the unconditional density of X. Assuming independence between X_i and $\hat{X}_i$:

$$\begin{aligned}\mathsf{E}[X_i \mid X = a] &= \int_0^a x_i f_a(x_i, a - x_i)dx_i \\ &= \frac{1}{f_X(a)}\int_0^a x_i f_i(x_i) f_{\hat{i}}(a - x_i)dx_i\end{aligned}$$

showing $\mathsf{E}[X_i \mid X = a]$ is the convolution of the functions $x_i \mapsto x_i f_i(x_i)$ and $f_{\hat{i}}$. The convolution can be computed using FFTs. In the case $f_X(a)$ is very small these estimates may be numerically unreliable.

- exlea_line $=\mathsf{E}[X_i \mid X \le a]$ is given by

$$\begin{aligned}\mathsf{E}[X_i \mid X \le a] &= \mathsf{E}[\mathsf{E}(X_i \mid X \le a] \mid X) \\ &= \int_0^a \mathsf{E}[X_i \mid X \le a, X = x] f_{\{X \mid X \le a\}}(x)dx \\ &= \frac{1}{F_X(a)} \int_0^a \mathsf{E}[X_i \mid X = x] f_X(x)dx\end{aligned}$$

can be computed for all a using the cumulative sums. Care is needed when a is so small that $F(a)$ is very small.

- exgta_line $= E(X_i \mid X \ge a)$ can be computed using $\mathsf{E}[X] = E(X_i \mid X \le a)F(a) + \mathsf{E}[X_i \mid X > a]S(a)$. It could also be computed with a reverse cumulative sum.
- exi_x_line $=\mathsf{E}[X_i/X]$, the unconditional average proportion of losses from line i is computed as

$$\begin{aligned}\mathsf{E}[X_i/X] &= \mathsf{E}_X[\mathsf{E}[X_i/X \mid X]] \\ &= \mathsf{E}_X[\mathsf{E}[X_i \mid X]/X] \\ &= \int_0^\infty \mathsf{E}[X_i \mid X = x]x^{-1} f_X(x)dx.\end{aligned}$$

- exi_xlea_line $=\mathsf{E}[X_i/X \mid X \le a]$ is computed using cumulative sums via

$$\mathsf{E}[X_i/X \mid X \le a] = \frac{1}{F(a)} \int_0^a \mathsf{E}[X_i \mid X = x]x^{-1} f_X(x)dx.$$

- exi_xgta_line $=\mathsf{E}[X_i/X \mid X > a] = \alpha_i(a)$ computed from $\mathsf{E}[X_i/X]$ and $\mathsf{E}[X_i/X \mid X \le a]$ as usual.
- exa_line $=\mathsf{E}[X_i(a)]$ is the loss cost for line i using the equal priority rule. It is computed by conditioning on X

$$\begin{aligned}\mathsf{E}[X_i(a)] &= \mathsf{E}[X_i(a] \mid X \le a)F(a) + \mathsf{E}[X_i(a] \mid X > a)S(a) \\ &= \mathsf{E}[X_i \mid X \le a]F(a) + a\mathsf{E}[X_i/X \mid X > a]S(a)\end{aligned}$$

showing it is a simple weighted average of $\mathsf{E}[X_i \mid X \le a]$ and $\mathsf{E}[X_i/X \mid X > a]$, both of which have already been computed. The computation could also be carried out using $\mathsf{E}[X_i; X \le a]$ and $\mathsf{E}[X_i/X; X > a]$ which would avoid multiplying and dividing by F and S.

- e2pri_line $=\mathsf{E}[X_{i,2}(a)]$ is the recovery to X_i when it is subordinate to $\hat{X}_i$ and total assets $= a$. It can also be computed using FFTs. Assuming independence between the lines the recovery to line i given $\hat{X}_i$ is

$$X_{i,2}(a, \hat{X}_i) = \max(0, \min(X_{i,2}, a - \hat{X}_i)) = X_{i,2} \wedge (a - \hat{X}_i)^+$$

which can be computed as

$$\begin{aligned}\mathsf{E}[X_{i,2}(a)] &= \mathsf{E}_{\hat{X}_i}[\mathsf{E}[X_{i,2}(a) \mid \hat{X}_i]] \\ &= \mathsf{E}_{\hat{X}_i}[\mathsf{E}[X_i \wedge (a - \hat{X}_i)^+ \mid \hat{X}_i]] \\ &= \int_0^a \mathsf{E}[X_i \wedge (a - x) \mid \hat{X}_i = x) f_{\hat{i}}(x)dx \\ &= \int_0^a \mathsf{E}[X_i \wedge (a - x)] f_{\hat{i}}(x)dx\end{aligned}$$

showing $\mathsf{E}[X_{i,2}(a)]$ is the convolution of the functions $x \mapsto \mathsf{E}[X_i \wedge x]$ and $f_{\hat{i}}$, i.e. of the limited expected values of X_i on a stand-alone basis and the density of $\hat{X}_i$.

- epd_i_line are the expected policyholder deficits of line with assets a. When $i = 1$ the computation is for the standalone line, when $i = 1$ for the line with equal priority and when $i = 2$ for the line with second priority relative to all other lines. The calculation are all simple

$$\begin{aligned}\text{epd}_0(X_i, a) &= \frac{\mathsf{E}[X_i] - \mathsf{E}[X_i \wedge a]}{\mathsf{E}[X_i]}\\ \text{epd}_1(X_i, a) &= \frac{\mathsf{E}[X_i] - \mathsf{E}[X_i(a)]}{\mathsf{E}[X_i]}\\ \text{epd}_2(X_i, a) &= \frac{\mathsf{E}[X_i] - \mathsf{E}[X_{i,2}(a)]}{\mathsf{E}[X_i]}\end{aligned}$$

The upshot of these calculations is that all the required values, for all levels of capital a can be computed in time $O(mn\log(n))$ where m is the number of lines of business and n is the length of the vector used to discretize the underlying distributions. Without using FFTs the calculations would take $O(mn^2)$. Since n is typically in the range 2^{10} to 2^{20} FFTs provide a huge speed-up. Using simple simulations would be completely impractical for the delicate calculations involved.

The calculation of $\mathsf{E}[X_i(a)] = \mathsf{E}[X_i \mid X \le a]F(a) + a\mathsf{E}[X_i/X \mid X > a]S(a)$ depends critically on the fact that the same values $\mathsf{E}[X_i \mid X = x]$ and $\mathsf{E}[X_i/X \mid X > a]$ are used for all values of a. Only the weights $F(a)$ and $S(a)$ change with a. As a result $\mathsf{E}[X_i(a)]$ can be computed in one sweep of length n. If different values were required for each value of a the complexity would jump up to $O(mn \times n^2)$ (or $O(mn \times n\log(n))$ if it is possible to use FFTs). This is unfortunately the situation when one line is collateralized because the ratio of capital to collateral determines the allocation of assets in insolvency.

Now we compute the impact of applying a distortion g to the underlying probabilities, i.e. discuss premium allocations.

Let E_g denote expected values with respect to the distorted probabilities defined by g.

Table 6: Variables and computational complexity by line i, with distorted probabilities. Complexity refers to additional complexity beyond values already computed.

Variable	Meaning	Computation	Complexity
gS, gF	$g(S(x))$ and $1 - g(S(x))$		$O(n)$
gp_total	Estimate of $-dg(S(x))/dx$	Difference of $g(S)$	$O(n)$
exag_total	$\mathsf{E}_g[X \wedge a]$	Cumulative sum of $g(S)$	$O(n)$
exag_line	$\mathsf{E}_g[X_i(a)]$	See below	$O(n)$

- exag_total is easy to compute as the cumulative sums of $g(S)$
- exag_line is computed as

$$\begin{aligned}\mathsf{E}_g[X_i(a)] &= \mathsf{E}\left[X_i\frac{X \wedge a}{X}g'S(X)\right]\\ &= \mathsf{E}\left[\mathsf{E}\left[X_i\frac{X \wedge a}{X}g'S(X) \mid X\right]\right]\\ &= \mathsf{E}\left[\mathsf{E}[X_i \mid X]1_{\{X\le a\}}g'S(X)\right] + a\mathsf{E}\left[\frac{\mathsf{E}[X_i \mid X]}{X}1_{\{X>a\}}g'S(X)\right]\\ &= \int_0^a \mathsf{E}[X_i \mid X = x]g'(S(x))f_X(x)dx + \int_a^\infty \mathsf{E}[X_i \mid X = x]x^{-1}g'S(x)f_X(x)dx.\end{aligned}$$

  The first integral is computed as a cumulative sum of its terms, the second is computed as a reverse cumulative sum, both using `exeqa`. This expectation can also be expressed using $\beta_i(a)$.

- If g has a probability mass at $s = 0$ then **how are the masses dealt with**?

Finally we discuss computing the impact of line specific collateral.

Computing the impact of collateral on recoveries. Computes the expected recoveries to line X_i when there are assets a but line i has collateral $c \le a$. This calculation, alas, cannot be performed quickly using FFTs. It has to be computed mirroring the three way split of the default zone: no default, default and line i just paid full collateral (which requires $X_i < cx/a$ where x is total loss), and line i is paid its usual pro rata proportion of assets.

5.8.4 `Distortion` Class Calculations

Todo: Documentation to follow.

5.8.5 `Bounds` Class Calculations

Todo: Documentation to follow.

5.9 Working With Samples

Objectives: Describe working with samples including the switcheroo trick, the Iman-Conover algorithm and the re-arrangement algorithm.

Audience: Advanced users and programmers.

Prerequisites: DecL, general use of `aggregate`, probability. Probability, measures of association, multivariate distributions, matrix algebra.

See also: 5_x_iman_conover, 5_x_rearrangement_algorithm, *Working With Samples*, *API Reference*, and *A Ten Minute Guide to aggregate*.

Contents:

5.9.1 Helpful References

- Conover [1999]
- Mildenhall [2005]
- Puccetti and Ruschendorf [2012]
- Embrechts *et al.* [2013]
- Mildenhall and Major [2022], Section 4.2.5.

5.9.2 Using Samples and The Switcheroo Trick

Todo: Documentation to follow.

5.9.3 The Iman-Conover Method

Basic Idea

Here is the basic idea of the Iman-Conover method. Given samples of n values from two known marginal distributions X and Y and a desired correlation ρ between them, re-order the samples to have the same rank order as a reference distribution, of size $n \times 2$, with linear correlation ρ. Since linear correlation and rank correlation are typically close, the re-ordered output will have approximately the desired correlation structure. What makes the IC method work so effectively is the existence of easy algorithms to determine samples from reference distributions with prescribed linear correlation structures.

Theoretical Derivation

Suppose that M is an n element sample from an r dimensional multivariate distribution, so M is an $n \times r$ matrix. Assume that the columns of M are uncorrelated, have mean zero, and standard deviation one. Let M' denote the transpose of M. These assumptions imply that the correlation matrix of the sample M can be computed as $n^{-1}\mathsf{M}'\mathsf{M}$, and because the columns are independent, $n^{-1}\mathsf{M}'\mathsf{M} = \mathsf{id}$. (There is no need to scale the covariance matrix by the row and column standard deviations because they are all one. In general $n^{-1}\mathsf{M}'\mathsf{M}$ is the covariance matrix of M.)

Let S be a correlation matrix, i.e. S is a positive semi-definite symmetric matrix with 1's on the diagonal and all elements ≤ 1 in absolute value. In order to rule out linearly dependent variables assume S is positive definite. These assumptions ensure S has a Choleski decomposition

$$\mathsf{S} = \mathsf{C}'\mathsf{C}$$

for some upper triangular matrix C, see Golub Golub or Press et al. Set $\mathsf{T} = \mathsf{MC}$. The columns of T still have mean zero, because they are linear combinations of the columns of M which have zero mean by assumption. It is less obvious, but still true, that the columns of T still have standard deviation one. To see why, remember that the covariance matrix of T is

$$n^{-1}\mathsf{T}'\mathsf{T} = n^{-1}\mathsf{C}'\mathsf{M}'\mathsf{MC} = \mathsf{C}'\mathsf{C} = \mathsf{S},$$

since $n^{-1}\mathsf{M}'\mathsf{M} = \mathsf{id}$ is the identity by assumption. Now S is actually the correlation matrix too because the diagonal is scaled to one, so the covariance and correlation matrices coincide. The process of converting M, which is easy to simulate, into T, which has the desired correlation structure S, is the theoretical basis of the IC method.

It is important to note that estimates of correlation matrices, depending on how they are constructed, need not have the mathematical properties of a correlation matrix. Therefore, when trying to use an estimate of a correlation matrix in an algorithm, such as the Iman-Conover, which actually requires a proper correlation matrix as input, it may be necessary to check the input matrix does have the correct mathematical properties.

Next we discuss how to make $n \times r$ matrices M, with independent, mean zero columns. The basic idea is to take n numbers $a_1, \dots, a_n$ with $\sum_i a_i = 0$ and $n^{-1}\sum_i a_i^2 = 1$, use them to form one $n \times 1$ column of M, and then to copy it r times. Finally randomly permute the entries in each column to make them independent as columns of random variables. Iman and Conover call the a_i "scores". They discuss several possible definitions for the scores, including scaled versions of $a_i = i$ (ranks) and a_i uniformly distributed. They note that the shape of the output multivariate distribution depends on the scores. All of the examples in their paper use normal scores. We will discuss normal scores here, and consider alternatives in Section 1.4.1.

Given that the scores will be based on normal random variables, we can either simulate n random standard normal variables and then shift and re-scale to ensure mean zero and standard deviation one, or we can use a stratified sample from the standard normal, $a_i = \Phi^{-1}(i/(n+1))$. By construction, the stratified sample has mean zero which is an advantage. Also, by symmetry, using the stratified sample halves the number of calls to Φ^{-1}. For these two reasons we prefer it in the algorithm below.

The correlation matrix of M, constructed by randomly permuting the scores in each column, will only be approximately equal to id because of random simulation error. In order to correct for the slight error which could be introduced Iman and Conover use another adjustment in their algorithm. Let $\mathsf{EE} = n^{-1}\mathsf{M}'\mathsf{M}$ be the actual correlation matrix of M and let $\mathsf{EE} = \mathsf{F}'\mathsf{F}$ be the Choleski decomposition of EE, and define $\mathsf{T} = \mathsf{MF}^{-1}\mathsf{C}$. The columns of

T have mean zero, and the covariance matrix of T is

$$\begin{aligned} n^{-1}\mathsf{T}'\mathsf{T} &= n^{-1}\mathsf{C}'\mathsf{F}'^{-1}\mathsf{M}'\mathsf{M}\mathsf{F}^{-1}\mathsf{C} \\ &= \mathsf{C}'\mathsf{F}'^{-1}\mathsf{E}\mathsf{E}\mathsf{F}^{-1}\mathsf{C} \\ &= \mathsf{C}'\mathsf{F}'^{-1}\mathsf{F}'\mathsf{F}\mathsf{F}^{-1}\mathsf{C} \\ &= \mathsf{C}'\mathsf{C} \\ &= \mathsf{S}, \end{aligned}$$

and hence T has correlation matrix exactly equal to S, as desired. If EE is singular then the column shuffle needs to be repeated.

Now the reference distribution T with exact correlation structure S is in hand, all that remains to complete the IC method is to re-order the each column of the input distribution X to have the same rank order as the corresponding column of T.

Algorithm

Here is a more algorithmic description of the IC method. The description uses normal scores and the Choleski method to determine the reference distribution. As we discussed above, it is possible to make other choices in place of these and they are discussed in Section 1.4. We will actually present two versions of the core algorithm. The first, called "Simple Algorithm" deals with the various matrix operations at a high level. The second "Detailed Algorithm" takes a more sophisticated approach to the matrix operations, including referencing appropriate Lapack routines. Lapack is a standard set of linear algebra functions. Software vendors provide very high performance implementations of Lapack, many of which are used in CPU benchmarks. Several free Windows implementations are available on the web. The reader should study the simple algorithm first to understand what is going in the IC method. In order to code a high performance implementation you should follow the steps outlined in the detailed algorithm. Both algorithms have the same inputs and outputs.

An $n \times r$ matrix X consisting of n samples from each of r marginal distributions, and a desired correlation matrix S.

The IC method does not address how the columns of X are determined. It is presumed that the reader has sampled from the appropriate distributions in some intelligent manner. The matrix S must be a correlation matrix for linearly independent random variables, so it must be symmetric and positive definite. If S is not symmetric positive semi-definite the algorithm will fail at the Choleski decomposition step. The output is a matrix T each of whose columns is a permutation of the corresponding column of X and whose approximate correlation matrix is S.

1. Make one column of scores $a_i = \Phi^{-1}(i/(n+1))$ for $i = 1, \dots, n$ and rescale to have standard deviation one.
2. Copy the scores r times to make the score matrix M.
3. Randomly permute the entries in each column of M.
4. Compute the correlation matrix $\mathsf{EE} = n^{-1}\mathsf{M}'\mathsf{M}$ of M.
5. Compute the Choleski decomposition $\mathsf{EE} = \mathsf{F}'\mathsf{F}$ of EE.
6. Compute the Choleski decomposition $\mathsf{S} = \mathsf{C}'\mathsf{C}$ of the desired correlation matrix S.
7. Compute $\mathsf{T} = \mathsf{MF}^{-1}\mathsf{C}$. The matrix T has exactly the desired correlation structure.
8. Let Y be the input matrix X with each column reordered to have exactly the same rank ordering as the corresponding column of T.
9. Compute the Choleski decomposition of S, $\mathsf{S} = \mathsf{C}'\mathsf{C}$, with C upper triangular. If the Choleski algorithm fails then S is not a valid correlation matrix. Flag an error and exit. Checking S is a correlation matrix in Step 1 avoids performing wasted calculations and allows the routine to exit as quickly as possible. Also check that all the diagonal entries of S are 1 so S has full rank. Again flag an error and exit if not. The Lapack routine `DPOTRF` can use be used to compute the Choleski decomposition. In the absence of Lapack, $\mathsf{C} = (c_{ij})$ can be computed recursively using

$$c_{ij} = \frac{s_{ij} - \sum_{k=1}^{j-1} c_{ik}c_{jk}}{\sqrt{1 - \sum_{k=1}^{j-1} c_{jk}^2}}$$

for $1 \le i \le j \le n$—since all the diagonal elements of S equal one. The empty sum $\sum_0^0 = 0$ and for $j > i$ the denominator equals c_{ii} and the elements of C should be calculated from left to right, top to bottom. See Wang or Herzog.

10. Let $m = \lfloor n/2 \rfloor$ be the largest integer less than or equal to $n/2$ and $v_i = \Phi^{-1}(i/(2m+1))$ for $i = 1, \dots, m$.
11. If n is odd set

$$\mathsf{v} = (v_m, v_{m-1}, \dots, v_1, 0, -v_1, \dots, -v_m)$$

and if n is even set

$$\mathsf{v} = (v_m, v_{m-1}, \dots, v_1, -v_1, \dots, -v_m).$$

Here we have chosen to use normal scores. Other distributions could be used in place of the normal, as discussed in Section 1.4.1. Also note that by taking advantage of the symmetry of the normal distribution halves the number of calls to Φ^{-1} which is relatively computationally expensive. If multiple calls will be made to the IC algorithm then store v for use in future calls.

12. Form the $n \times r$ score matrix M from r copies of the scores vector v.
13. Compute $m_{xx} = n^{-1} \sum_i v_i^2$, the variance of v. Note that $\sum_i v_i = 0$ by construction.
14. Randomly shuffle columns $2, \dots, r$ of the score matrix.
15. Compute the correlation matrix EE of the shuffled score matrix M. Each column of M has mean zero, by construction, and variance m_{xx}. The correlation matrix is obtained by dividing each element of $\mathsf{M'M}$ by m_{xx}. The matrix product can be computed using the Lapack routine `DGEMM`. If EE is singular repeat step 6.
16. Determine Choleski decomposition $\mathsf{EE} = \mathsf{F'F}$ of EE using the Lapack routine `DPOTRF`. Because EE is a correlation matrix it must be symmetric and positive definite and so is guaranteed to have a Choleski root.
17. Compute $\mathsf{F}^{-1}\mathsf{C}$ using the Lapack routine `DTRTRS` to solve the linear equation $\mathsf{FA} = \mathsf{C}$ for A. Solving the linear equation avoids a time consuming matrix inversion and multiplication. The routine `DTRTRS` is optimized for upper triangular input matrices.
18. Compute the correlated scores $\mathsf{T} = \mathsf{MF}^{-1}\mathsf{C} = \mathsf{MA}$ using `DGEMM`. The matrix T has exactly the desired correlation structure.
19. Compute the ranks of the elements of T. Ranks are computed by indexing the columns of T as described in Section 8.4 of Press et al. Let $r(k)$ denote the index of the kth ranked element of T.
20. Let Y be the $n \times r$ matrix with ith column equal to the ith column of the input matrix X given the same rank order as T. The re-ordering is performed using the ranks computed in the previous step. First sort the input columns into ascending order if they are not already sorted and then set $\mathsf{Y}_{i,k} = \mathsf{X}_{i,r(k)}$.

The output of the algorithm is a matrix Y each of whose columns is a permutation of the corresponding column of the input matrix X. The rank correlation matrix of Y is identical to that of a multivariate distribution with correlation matrix S.

Simple Example of Iman-Conover

Having explained the IC method, we now give a simple example to explicitly show all the details. The example will work with $n = 20$ samples and $r = 4$ different marginals. The marginals are samples from four lognormal distributions, with parameters $\mu = 12, 11, 10, 10$ and $\sigma = 0.15, 0.25, 0.35, 0.25$. The input matrix is

$$\mathsf{X} = \begin{pmatrix} 123,567 & 44,770 & 15,934 & 13,273 \\ 126,109 & 45,191 & 16,839 & 15,406 \\ 138,713 & 47,453 & 17,233 & 16,706 \\ 139,016 & 47,941 & 17,265 & 16,891 \\ 152,213 & 49,345 & 17,620 & 18,821 \\ 153,224 & 49,420 & 17,859 & 19,569 \\ 153,407 & 50,686 & 20,804 & 20,166 \\ 155,716 & 52,931 & 21,110 & 20,796 \\ 155,780 & 54,010 & 22,728 & 20,968 \\ 161,678 & 57,346 & 24,072 & 21,178 \\ 161,805 & 57,685 & 25,198 & 23,236 \\ 167,447 & 57,698 & 25,393 & 23,375 \\ 170,737 & 58,380 & 30,357 & 24,019 \\ 171,592 & 60,948 & 30,779 & 24,785 \\ 178,881 & 66,972 & 32,634 & 25,000 \\ 181,678 & 68,053 & 33,117 & 26,754 \\ 184,381 & 70,592 & 35,248 & 27,079 \\ 206,940 & 72,243 & 36,656 & 30,136 \\ 217,092 & 86,685 & 38,483 & 30,757 \\ 240,935 & 87,138 & 39,483 & 35,108 \end{pmatrix}.$$

Note that the marginals are all sorted in ascending order. The algorithm does not actually require pre-sorting the marginals but it simplifies the last step.

The desired target correlation matrix is

$$\mathsf{S} = \begin{pmatrix} 1.000 & 0.800 & 0.400 & 0.000 \\ 0.800 & 1.000 & 0.300 & -0.200 \\ 0.400 & 0.300 & 1.000 & 0.100 \\ 0.000 & -0.200 & 0.100 & 1.000 \end{pmatrix}.$$

The Choleski decomposition of S is

$$\mathsf{C} = \begin{pmatrix} 1.000 & 0.800 & 0.400 & 0.000 \\ 0.000 & 0.600 & -0.033 & -0.333 \\ 0.000 & 0.000 & 0.916 & 0.097 \\ 0.000 & 0.000 & 0.000 & 0.938 \end{pmatrix}.$$

Now we make the score matrix. The basic scores are $\Phi^{-1}(i/21)$, for $i = 1, \dots, 20$. We scale these by 0.868674836252965 to get a vector v with standard deviation one. Then we combine four v's and shuffle randomly to get

$$
\mathsf{M} = \begin{pmatrix}
-1.92062 & 1.22896 & -1.00860 & -0.49584 \\
-1.50709 & -1.50709 & -1.50709 & 0.82015 \\
-1.22896 & 1.92062 & 0.82015 & -0.65151 \\
-1.00860 & -0.20723 & 1.00860 & -1.00860 \\
-0.82015 & 0.82015 & 0.34878 & 1.92062 \\
-0.65151 & -1.22896 & -0.65151 & 0.20723 \\
-0.49584 & -0.65151 & 1.22896 & -0.34878 \\
-0.34878 & -0.49584 & -0.49584 & -0.06874 \\
-0.20723 & -1.00860 & 0.20723 & 0.65151 \\
-0.06874 & 0.49584 & 0.06874 & -1.22896 \\
0.06874 & -0.34878 & -1.22896 & 0.49584 \\
0.20723 & 0.34878 & 0.65151 & 0.34878 \\
0.34878 & -0.06874 & -0.20723 & 1.22896 \\
0.49584 & -1.92062 & -0.82015 & -0.20723 \\
0.65151 & 0.20723 & 1.92062 & -1.92062 \\
0.82015 & 1.00860 & 1.50709 & 1.50709 \\
1.00860 & -0.82015 & -1.92062 & 1.00860 \\
1.22896 & 1.50709 & 0.49584 & -1.50709 \\
1.50709 & 0.06874 & -0.06874 & 0.06874 \\
1.92062 & 0.65151 & -0.34878 & -0.82015
\end{pmatrix}.
$$

As described in Section 1.1, M is approximately independent. In fact M has covariance matrix

$$
\mathsf{EE} = \begin{pmatrix}
1.0000 & 0.0486 & 0.0898 & -0.0960 \\
0.0486 & 1.0000 & 0.4504 & -0.2408 \\
0.0898 & 0.4504 & 1.0000 & -0.3192 \\
-0.0960 & -0.2408 & -0.3192 & 1.0000
\end{pmatrix}
$$

and EE has Choleski decomposition

$$
\mathsf{F} = \begin{pmatrix}
1.0000 & 0.0486 & 0.0898 & -0.0960 \\
0.0000 & 0.9988 & 0.4466 & -0.2364 \\
0.0000 & 0.0000 & 0.8902 & -0.2303 \\
0.0000 & 0.0000 & 0.0000 & 0.9391
\end{pmatrix}.
$$

Thus $\mathsf{T} = \mathsf{MF}^{-1}\mathsf{C}$ is given by

$$
\mathsf{T} = \begin{pmatrix}
-1.92062 & -0.74213 & -2.28105 & -1.33232 \\
-1.50709 & -2.06697 & -1.30678 & 0.54577 \\
-1.22896 & 0.20646 & -0.51141 & -0.94465 \\
-1.00860 & -0.90190 & 0.80546 & -0.65873 \\
-0.82015 & -0.13949 & -0.31782 & 1.76960 \\
-0.65151 & -1.24043 & -0.27999 & 0.23988 \\
-0.49584 & -0.77356 & 1.42145 & 0.23611 \\
-0.34878 & -0.56670 & -0.38117 & -0.14744 \\
-0.20723 & -0.76560 & 0.64214 & 0.97494 \\
-0.06874 & 0.24487 & -0.19673 & -1.33695 \\
0.06874 & -0.15653 & -1.06954 & 0.14015 \\
0.20723 & 0.36925 & 0.56694 & 0.51206 \\
0.34878 & 0.22754 & -0.06362 & 1.19551 \\
0.49584 & -0.77154 & 0.26828 & 0.03168 \\
0.65151 & 0.62666 & 2.08987 & -1.21744 \\
0.82015 & 1.23804 & 1.32493 & 1.85680 \\
1.00860 & 0.28474 & -1.23688 & 0.59246 \\
1.22896 & 1.85260 & 0.17411 & -1.62428 \\
1.50709 & 1.20294 & 0.39517 & 0.13931 \\
1.92062 & 1.87175 & -0.04335 & -0.97245
\end{pmatrix}.
$$

An easy calculation will verify that T has correlation matrix S, as required.

To complete the IC method we must re-order each column of X to have the same rank order as T. The first column does not change because it is already in ascending order. In the second column, the first element of Y must be the 14th element of X, the second the 20th, third 10th and so on. The ranks of the other elements are the transpose of

$$\begin{pmatrix} 14 & 20 & 10 & 18 & 11 & 19 & 17 & 13 & 15 & 8 \\ 12 & 6 & 9 & 16 & 5 & 3 & 7 & 2 & 4 & 1 \\ 20 & 19 & 16 & 4 & 14 & 13 & 2 & 15 & 5 & 12 \\ 17 & 6 & 11 & 8 & 1 & 3 & 18 & 9 & 7 & 10 \\ 18 & 6 & 15 & 14 & 2 & 8 & 9 & 13 & 4 & 19 \\ 10 & 7 & 3 & 12 & 17 & 1 & 5 & 20 & 11 & 16 \end{pmatrix}$$

and the resulting re-ordering of X is

$$\mathsf{T} = \begin{pmatrix} 123,567 & 50,686 & 15,934 & 16,706 \\ 126,109 & 44,770 & 16,839 & 25,000 \\ 138,713 & 57,685 & 17,620 & 19,569 \\ 139,016 & 47,453 & 35,248 & 20,166 \\ 152,213 & 57,346 & 20,804 & 30,757 \\ 153,224 & 45,191 & 21,110 & 24,019 \\ 153,407 & 47,941 & 38,483 & 23,375 \\ 155,716 & 52,931 & 17,859 & 20,796 \\ 155,780 & 49,420 & 33,117 & 27,079 \\ 161,678 & 58,380 & 22,728 & 15,406 \\ 161,805 & 54,010 & 17,265 & 23,236 \\ 167,447 & 66,972 & 32,634 & 24,785 \\ 170,737 & 57,698 & 24,072 & 30,136 \\ 171,592 & 49,345 & 30,357 & 20,968 \\ 178,881 & 68,053 & 39,483 & 16,891 \\ 181,678 & 72,243 & 36,656 & 35,108 \\ 184,381 & 60,948 & 17,233 & 26,754 \\ 206,940 & 86,685 & 25,393 & 13,273 \\ 217,092 & 70,592 & 30,779 & 21,178 \\ 240,935 & 87,138 & 25,198 & 18,821 \end{pmatrix}.$$

The rank correlation matrix of Y is exactly S. The actual linear correlation is only approximately equal to S. The achieved value is

$$\begin{pmatrix} 1.00 & 0.85 & 0.26 & -0.11 \\ 0.85 & 1.00 & 0.19 & -0.20 \\ 0.26 & 0.19 & 1.00 & 0.10 \\ -0.11 & -0.20 & 0.10 & 1.00 \end{pmatrix},$$

a fairly creditable performance given the input correlation matrix and the very small number of samples $n = 20$. When used with larger sized samples the IC method typically produces a very close approximation to the required correlation matrix, especially when the marginal distributions are reasonably symmetric.

Extensions of Iman-Conover

Following through the explanation of the IC method shows that it relies on a choice of multivariate reference distribution. A straightforward method to compute a reference is to use the Choleski decomposition method Equation ([coolA]) applied to certain independent scores. The example in Section 1.3 used normal scores. However nothing prevents us from using other distributions for the scores provided they are suitably normalized to have mean zero and standard deviation one.

Another approach to IC is to use a completely different multivariate distribution as reference. There are several other families of multivariate distributions, including the elliptically contoured distribution family (which includes the normal and t as a special cases) and multivariate Laplace distribution, which are easy to simulate from. Changing scores is actually an example of changing the reference distribution; however, for the examples we consider the exact form of the new reference is unknown.

Alternative Scores

The choice of score distribution has a profound effect on the multivariate distribution output by the IC method. The basic Iman-Conover algorithm uses normally distributed scores. We now show the impact of using exponentially and uniformly distributed scores.

The next figure shows three bivariate distributions with identical marginal distributions (shown in the lower right hand plot), the same correlation coefficient of 0.643 ± 0.003 but using normal scores (top left), exponential scores (top rigtht) and uniform scores (lower left). The input correlation to the IC method was 0.65 in all three cases and there are 1000 pairs in each plot. Here the IC method produced bivariate distributions with actual correlation coefficient extremely close to the requested value.

The normal scores produce the most natural looking bivariate distribution, with approximately elliptical contours. The bivariate distributions with uniform or exponential scores look unnatural, but it is important to remember that if all you know about the bivariate distribution are the marginals and correlation coefficient all three outcomes are possible.

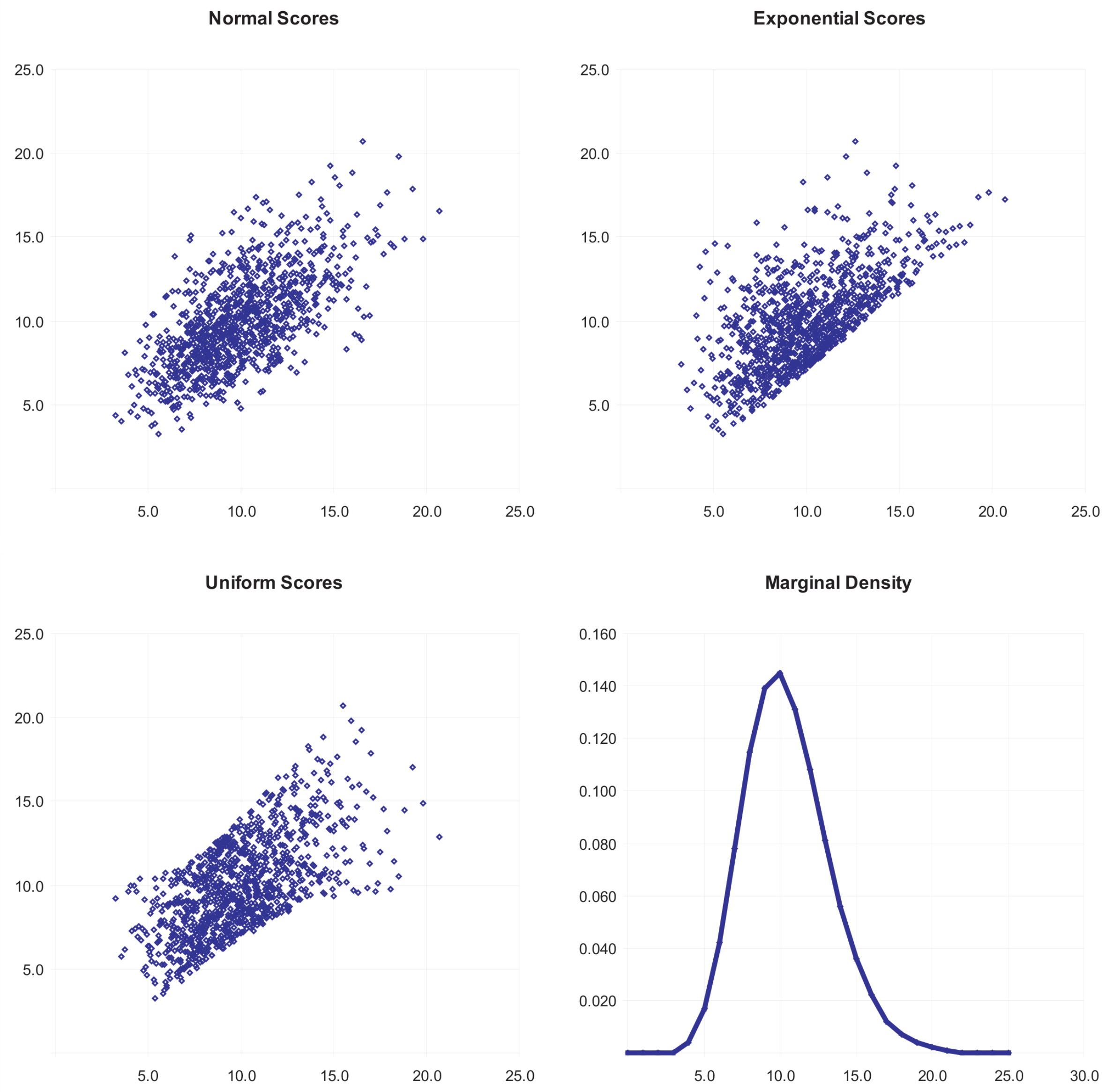


Fig. 1: Bivariate distributions with normal, uniform and exponential scores.

Figure MISSING shows the distribution of the sum of the two marginals for each of the three bivariate distributions and for independent marginals. The sum with exponential scores has a higher kurtosis (is more peaked) than with normal scores. As expected all three dependent sums have visibly thicker tails than the independent sum.

Iman and Conover considered various different score distributions in their paper. They preferred normal scores as giving more natural looking, elliptical contours. Certainly, the contours produced using exponential or uniform scores appear unnatural. If nothing else they provide a sobering reminder that knowing the marginal distributions and correlation coefficient of a bivariate distribution does not come close to fully specifying it!

Multivariate Reference Distributions

The IC method needs some reference multivariate distribution to determine an appropriate rank ordering for the input marginals. So far we have discussed using the Choleski decomposition trick in order to determine a multivariate normal reference distribution. However, any distribution can be used as reference provided it has the desired correlation structure. Multivariate distributions that are closely related by formula to the multivariate normal, such as elliptically contoured distributions and asymmetric Laplace distributions, can be simulated using the Choleski trick.

Elliptically contoured distributions are a family which extends the normal. For a more detailed discussion see Fang and Zhang. The multivariate t-distribution and symmetric Laplace distributions are in the elliptically contoured family. Elliptically contoured distributions must have characteristic equations of the form

$$\Phi(\mathsf{t}) = \exp(i\mathsf{t}'\mathsf{m})\phi(\mathsf{t}'\mathsf{S}\mathsf{t})$$

for some $\phi : \mathsf{R} \to \mathsf{R}$, where m is an $r \times 1$ vector of means and S is a $r \times r$ covariance matrix (nonnegative definite and symmetric). In one dimension the elliptically contoured distributions coincide with the symmetric distributions. The covariance is S, if it is defined.

If S has rank r then an elliptically contoured distribution x has a stochastic representation

$$\mathsf{x} = \mathsf{m} + R\mathsf{T}'\mathsf{u}^{(r)}$$

where T is the Choleski decomposition of S, so $\mathsf{S} = \mathsf{T}'\mathsf{T}$, $\mathsf{u}^{(r)}$ is a uniform distribution on the sphere in R^r, and R is a scale factor independent of $\mathsf{u}^{(r)}$. The idea here should be clear: pick a direction on the sphere, adjust by T, scale by a distance R and finally translate by the means m. A uniform distribution on a sphere can be created as $\mathsf{x}/\|\mathsf{x}\|$ where x has a multivariate normal distribution with identity covariance matrix. (By definition, $\|\mathsf{x}\|^2 = \sum_i x_i^2$ has a χ^2_r distribution.) Uniform vectors $\mathsf{u}^{(r)}$ can also be created by applying a random orthogonal matrix to a fixed vector $(1, 0, \dots, 0)$ on the sphere. Diaconis describes a method for producing random orthogonal matrices.

The t-copula with ν degrees of freedom has a stochastic representation

$$\mathsf{x} = \mathsf{m} + \frac{\sqrt{\nu}}{\sqrt{S}}\mathsf{z}$$

where $S \sim \chi^2_\nu$ and z is multivariate normal with means zero and covariance matrix S. Thus one can easily simulate from the multivariate t by first simulating multivariate normals and then simulating an independent S and multiplying.

The multivariate Laplace distribution is discussed in Kotz, Kozubowski and Podgorski. It comes in two flavors: symmetric and asymmetric. The symmetric distribution is also an elliptically contoured distribution. It has characteristic function of the form

$$\Phi(\mathsf{t}) = \frac{1}{1 + \mathsf{t}'\mathsf{S}\mathsf{t}/2}$$

where S is the covariance matrix. To simulate, use the fact that $\sqrt{W}\mathsf{X}$ has a symmetric Laplace distribution if W is exponential and X a multivariate normal with covariance matrix S.

The multivariate asymmetric Laplace distribution has characteristic function

$$\Psi(\mathsf{t}) = \frac{1}{1 + \mathsf{t}'\mathsf{S}\mathsf{t}/2 - i\mathsf{m}'\mathsf{t}}.$$

To simulate from WHAT, use the fact that

$$\mathsf{m}W + \sqrt{W}\mathsf{X}$$

has a symmetric Laplace distribution if W is exponential and X a multivariate normal with covariance matrix S and means zero. The asymmetric Laplace is not an elliptically contoured distribution.

The next figure compares IC samples produced using a normal copula to those produced with a t-copula. In both cases the marginals are normally distributed with mean zero and unit standard deviation. The t-copula has $\nu = 2$ degrees of freedom. In both figures the marginals are uncorrelated, but in the right the marginals are not independent. The t-copula has pinched tails, similar to Venter's Heavy Right Tailed copulas.

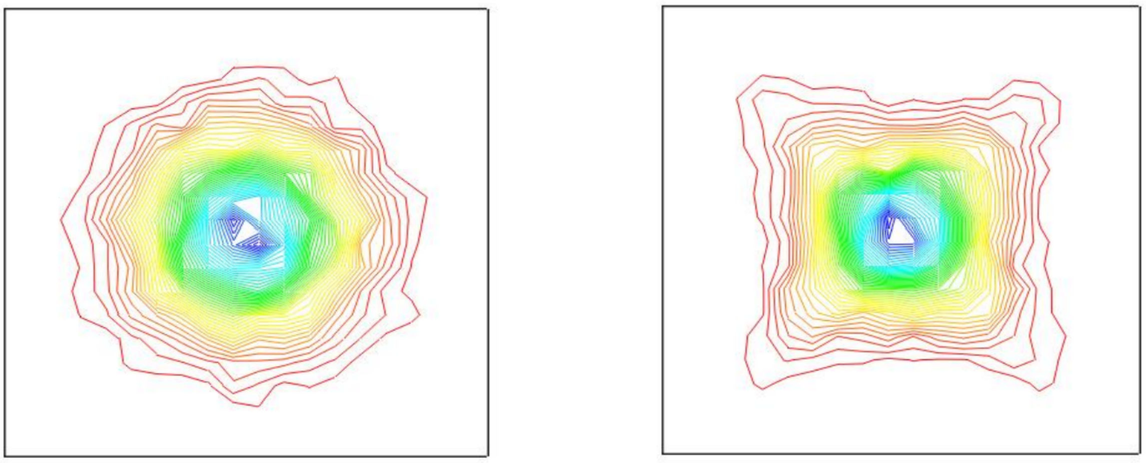


Fig. 2: IC samples produced from the same marginal and correlation matrix using the normal and t copula reference distributions.

Algorithms for Extended Methods

In Section 1.4.2 we described how the IC method can be extended by using different reference multivariate distributions. It is easy to change the IC algorithm to incorporate different reference distributions for t-copulas and asymmetric Laplace distributions. Follow the detailed algorithm to step 10. Then use the stochastic representation to simulate from the scaling distribution for each row and multiply each component by the resulting number, resulting in an adjusted T matrix. Then complete steps 11 and 12 of the detailed algorithm.

Comparison With the Normal Copula Method

By the normal copula method we mean the following algorithm, described in Wang or Herzog.

A set of correlated risks $(X_1, \dots, X_r)$ with marginal cumulative distribution functions F_i and Kendall's tau $\tau_{ij} = \tau(X_i, X_j)$ or rank correlation coefficients $r(X_i, X_j)$.

1. Convert Kendall's tau or rank correlation coefficient to correlation using

$$\rho_{ij} = \sin(\pi\tau_{ij}/2) = 2\sin(\pi r_{ij}/6)$$

   and construct the Choleski decomposition $\mathsf{S} = \mathsf{C}'\mathsf{C}$ of $\mathsf{S} = (\rho_{ij})$.

2. Generate r standard normal variables $\mathsf{Y} = (Y_1, \dots, Y_r)$.
3. Set $\mathsf{Z} = \mathsf{YC}$.
4. Set $u_i = \Phi(Z_i)$ for $i = 1, \dots, r$.
5. Set $X_i = F_i^{-1}(u_i)$.

The vectors $(X_1, \dots, X_r)$ form a sample from a multivariate distribution with prescribed correlation structure and marginals F_i.

The Normal Copula method works because of the following theorem from Wang.

[wangThm] Assume that $(Z_1, \dots, Z_k)$ have a multivariate normal joint probability density function given by

$$f(z_1, \dots, z_k) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} \exp(-\mathsf{z}'\Sigma^{-1}\mathsf{z}/2),$$

$\mathsf{z} = (z_1, \dots, z_k)$, with correlation coefficients $\Sigma_{ij} = \rho_{ij} = \rho(Z_i, Z_j)$. Let $H(z_1, \dots, z_k)$ be their joint cumulative distribution function. Then

$$C(u_1, \dots, u_k) = H(\Phi^{-1}(u_1), \dots, \Phi^{-1}(u_k))$$

defines a multivariate uniform cumulative distribution function called the normal copula.

For any set of given marginal cumulative distribution functions $F_1, \dots, F_k$, the set of variables

$$X_1 = F_1^{-1}(\Phi(Z_1)), \dots, X_k = F_1^{-1}(\Phi(Z_k))$$

have a joint cumulative function

$$F_{X_1,\dots,X_k}(x_1, \dots, x_k) = H(\Phi^{-1}(F_x(u_1)), \dots, \Phi^{-1}(F_k(u_k))$$

with marginal cumulative distribution functions $F_1, \dots, F_k$. The multivariate variables $(X_1, \dots, X_k)$ have Kendall's tau

$$\tau(X_i, X_j) = \tau(Z_i, Z_j) = \frac{2}{\pi} \arcsin(\rho_{ij})$$

and Spearman's rank correlation coefficients

$$\text{rkCorr}(X_i, X_j) = \text{rkCorr}(Z_i, Z_j) = \frac{6}{\pi} \arcsin(\rho_{ij}/2)$$

In the normal copula method we simulate from H and then invert using ([ncm]). In the IC method with normal scores we produce a sample from H such that $\Phi(z_i)$ are equally spaced between zero and one and then, rather than invert the distribution functions, we make the jth order statistic from the input sample correspond to $\Phi(z) = j/(n+1)$ where the input has n observations. Because the jth order statistic of a sample of n observations from a distribution F approximates $F^{-1}(j/(n+1))$ we see the normal copula and IC methods are doing essentially the same thing.

While the normal copula method and the IC method are confusingly similar there are some important differences to bear in mind. Comparing and contrasting the two methods should help clarify how the two algorithms are different.

1. Wang [1998] shows the normal copula method corresponds to the IC method when the latter is computed using normal scores and the Choleski trick.
2. The IC method works on a given sample of marginal distributions. The normal copula method generates the sample by inverting the distribution function of each marginal as part of the simulation process.
3. Though the use of scores the IC method relies on a stratified sample of normal variables. The normal copula method could use a similar method, or it could sample randomly from the base normals. Conversely a sample could be used in the IC method.
4. Only the IC method has an adjustment to ensure that the reference multivariate distribution has exactly the required correlation structure.
5. IC method samples have rank correlation exactly equal to a sample from a reference distribution with the correct linear correlation. Normal copula samples have approximately correct linear and rank correlations.
6. An IC method sample must be taken in its entirety to be used correctly. The number of output points is fixed by the number of input points, and the sample is computed in its entirety in one step. Some IC tools (@Risk, SCARE) produce output which is in a particular order. Thus, if you sample the nth observation from multiple simulations, or take the first n samples, you will not get a random sample from the desired distribution. However, if you select random rows from multiple simulations (or, equivalently, if you randomly permute the rows output prior to selecting the nth) then you will obtain the desired random sample. It is important to be aware of these issues before using canned software routines.
7. The normal copula method produces simulations one at a time, and at each iteration the resulting sample is a sample from the required multivariate distribution. That is, output from the algorithm can be partitioned and used in pieces.

In summary remember these differences can have material practical consequences and it is important not to misuse IC method samples.

Theoretical Underpinnings of the Iman-Conover Method

The theoretical foundations of the Iman-Conover method are elegantly justified by Vitale's Theorem Vitale [1990]. We will state Vitale's theorem, explain its relationship to the IC method, and sketch the proof. The result should give a level of comfort to practitioners using a simulation approach to modeling multivariate distributions. It is not necessary to follow the details laid out here in order to understand and use the IC method, so the uninterested reader can skip the rest of the section. The presentation we give follows Vitale's original paper Vitale [1990] closely.

Functional dependence and independence between two random variables are clearly opposite ends of the dependence spectrum. It is therefore surprising that Vitale's Theorem says that any bivariate distribution (U, V) can be approximated arbitrarily closely by a functionally dependent pair (U, TU) for a suitable transformation T.

In order to explain the set up of Vitale's theorem we need to introduce some notation. Let n be a power of 2. An interval of the form $((j-1)/n, j/n)$ for some $n \ge 1$ and $1 \le j \le n$ is called a dyadic interval of rank n. An invertible (Borel) measure-preserving map which maps by translation on each dyadic interval of rank n is called a permutation of rank n. Such a T just permutes the dyadic intervals, so there is a natural correspondence between permutations of n elements and transformations T. If the permutation of dyadic intervals has a single cycle (has order n in the symmetric group) then T is called a cyclic permutation.

Theorem. (Vitale) Let U and V be uniformly distributed variables. There is a sequence of cyclic permutations $T_1, T_2, \dots$ such that $(U, T_n U)$ converges in distribution to (U, V) as $n \to \infty$.

Recall convergence in distribution means that the distribution function of $(U, T_n U)$ tends to that of (U, V) at all points of continuity as $n \to \infty$.

The proof of Vitale's theorem is quite instructive and so we give a detailed sketch.

The proof is in two parts. The first constructs a sequence of arbitrary permutations T_n with the desired property. The second part shows it can be approximated with cyclic permutations. We skip the second refinement.

Divide the square $[0,1] \times [0,1]$ into sub-squares. We will find a permutation T such that the distributions of (U, V) and (U, TU) coincide on sub-squares. Reducing the size of the sub-squares will prove the result.

Fix n, a power of two. Let $I_j = ((j-1)/n, j/n)$, $j = 1, \dots, n$. We will find an invertible permutation T such that

$$\Pr(U \in I_j, TU \in I_k) = \Pr(U \in I_j, V \in I_k) := p_{jk}$$

for $j, k = 1, \dots, n$. Define

$$\begin{aligned}
I_{j1} &= ((j-1)/n, (j-1)/n + p_{j1}) \\
I_{j2} &= ((j-1)/n + p_{j1}, (j-1)/n + p_{j1} + p_{j2}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \dots \\
I_{jn} &= ((j-1)/n + p_{j1} + \dots + p_{j,n-1}, j/n)
\end{aligned}$$

and

$$\begin{aligned}
\tilde{I}_{j1} &= ((j-1)/n, (j-1)/n + p_{1j}) \\
\tilde{I}_{j2} &= ((j-1)/n + p_{1j}, (j-1)/n + p_{1j} + p_{2j}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \dots \\
\tilde{I}_{jn} &= ((j-1)/n + p_{1j} + \dots + p_{n-1,j}, j/n).
\end{aligned}$$

By construction the measure of I_{jk} equals the measure of $\tilde{I}_{kj}$. The invertible map T which sends each I_{jk} to $\tilde{I}_{kj}$ by translation is the map we need because

$$\begin{aligned}
\Pr(U \in I_j, T(U) \in I_k) &= \Pr(U \in I_j, U \in T^{-1}(I_k)) \\
&= \Pr(U \in I_j \cap T^{-1}(\bigcup_l \tilde{I}_{kl})) \\
&= \Pr(U \in \bigcup_l I_j \cap I_{lk}) \\
&= \Pr(U \in I_{jk}) \\
&= p_{jk},
\end{aligned}$$

since the only I_{lk} which intersects I_j is I_{jk} by construction, and U is uniform. The transformation T is illustrated schematically in Table 1 for $n = 3$. The fact 3 is not a power of 2 does not invalidate the schematic!

	$\tilde{I}_{33}$									p_{33}
I_3	$\tilde{I}_{32}$						p_{23}			
	$\tilde{I}_{31}$			p_{13}						
	$\tilde{I}_{23}$								p_{32}	
I_2	$\tilde{I}_{22}$					p_{22}				
	$\tilde{I}_{21}$		p_{12}							
	$\tilde{I}_{13}$							p_{31}		
I_1	$\tilde{I}_{12}$				p_{21}					
	$\tilde{I}_{11}$	p_{11}								
		I_{11}	I_{12}	I_{13}	I_{21}	I_{22}	I_{13}	I_{31}	I_{32}	I_{33}
			I_1			I_2			I_3	

If each p_{jk} is a dyadic rational then T is a permutation of the interval. If not then we approximate and use some more heavy duty results (a 1946 theorem of Birkhoff on representation by convex combinations of permutation matrices) to complete the proof.

Vitale's theorem can be extended to non-uniform distributions.

Corollary. (Vitale) Let U and V be arbitrary random variables. There is a sequence of functions $S_1, S_2, \dots$ such that $(U, S_n U)$ converges in distribution to (U, V) as $n \to \infty$.

Let F be the distribution function of U and G for V. Then $F(U)$ and $G(V)$ are uniformly distributed. Apply Vitale's theorem to get a sequence of functions T_n. Then $S_n = G^{-1} T_n F$ is the required transformation.

5.9.4 The Rearrangement Algorithm

The Rearrangement Algorithm (RA) is a practical and straightforward method to determine the worst-VaR sum. The RA works by iteratively making each marginal crossed (counter-monotonic) with the sum of the other marginal distributions. It is easy to program and suitable for problems involving hundreds of variables and millions of simulations.

The Rearrangement Algorithm was introduced in Puccetti and Ruschendorf [2012] and subsequently improved in Embrechts *et al.* [2013].

Algorithm Input: Input samples are arranged in a matrix $\tilde{X} = (x_{ij})$ with $i = 1, \dots, M$ rows corresponding to the simulations and $j = 1, \dots, d$ columns corresponding to the different marginals. VaR probability parameter p. Accuracy threshold $\epsilon > 0$ specifies convergence criterion.

Algorithm Steps

1. **Sort** each column of $\tilde{X}$ in descending order.
2. **Set** $N := \lceil (1-p)M \rceil$.
3. **Create** matrix X as the $N \times d$ submatrix of the top N rows of $\tilde{X}$.
4. **Randomly permute** rows within each column of X.
5. **Do Loop**
    - **Create** a new matrix Y as follows. **For** column $j = 1, \dots, d$:
        - **Create** a temporary matrix V_j by deleting the jth column of X
        - **Create** a column vector v whose ith element equals the sum of the elements in the ith row of V_j
        - **Set** the jth column of Y equal to the jth column of X arranged to have the opposite order to v, i.e., the largest element in the jth column of X is placed in the row of Y corresponding to the smallest element in v, the second largest with second smallest, etc.
    - **Compute** y, the $N \times 1$ vector with ith element equal to the sum of the elements in the ith row of Y.
    - **Compute** x from X similarly.

- **Compute** $y^* := \min(y)$, the smallest element of y.
- **Compute** $x^* := \min(x)$.
- **If** $y^* - x^* \ge \epsilon$ **then** set $X := Y$ and **repeat** the loop.
- **If** $y^* - x^* < \epsilon$ **then break** from the loop.

6. **Stack** Y on top of the $(M - N) \times d$ submatrix of $M - N$ bottom rows of $\tilde{X}$.
7. **Output**: The result approximates the worst VaR_p arrangement of $\tilde{X}$.

Only the top N values need special treatment; all the smaller values can be combined arbitrarily because they aren't included in the worst-VaR rearrangement. Given that X consists of the worst $1 - p$ proportion of each marginal, the required estimated VaR_p is the least row sum of Y, that is y^*. In implementation, x^* can be carried forward as the y^* from the previous iteration and not recomputed. The statistics x^* and y^* can be replaced with the variance of the row-sums of X and Y and yield essentially the same results.

Embrechts *et al.* [2013] report that while there is no analytic proof the algorithm always works, it performs very well based on examples and tests where we can compute the answer analytically.

Worked Example

Setup. Compute the worst $\mathsf{VaR}_{0.99}$ of the sum of lognormal distributions with mean 10 and coefficient of variations 1, 2, and 3 by applying the Rearrangement Algorithm to a stratified sample of $N = 40$ observations at and above the 99th percentile for the matrix X.

Solution. The table below shows the input and output of the Rearrangement Algorithm.

Table 7: Starting X is shown in the first three columns x_0, x_1, x_2. The column Sum shows the row sums $x_0 + x_1 + x_2$ corresponding to a comonotonic ordering. These four columns are all sorted in ascending order. The right-hand three columns, s_0, s_1, s_2 are the output, with row sum given in the Max VaR column. The worst-case VaR$_{0.99}$ is the minimum of the last column, 352.8. It is 45 percent greater than the additive VaR of 242.5. Only a sample from each marginal's largest 1 percent values is shown since smaller values are irrelevant to the calculation.

x_0	x_1	x_2	Sum	s_0	s_1	s_2	Max VaR
49.0	85.6	107.9	242.5	87.1	124.6	141.1	352.8
49.4	86.6	109.5	245.6	70.8	113.6	169.3	353.7
49.9	87.7	111.2	248.8	98.8	127.9	127.4	354.1
50.3	88.9	112.9	252.1	79.9	118.8	155.5	354.1
50.7	90.0	114.7	255.5	83.1	107.1	164.3	354.5
51.2	91.3	116.6	259.1	92.1	139.7	122.8	354.6
51.6	92.6	118.6	262.8	67.7	135.4	151.5	354.7
52.1	93.9	120.6	266.6	108.8	116.1	129.8	354.7
52.6	95.3	122.8	270.7	62.8	105.1	186.9	354.8
53.2	96.7	125.0	274.9	63.9	170.6	120.6	355.0
53.7	98.3	127.4	279.3	69.2	111.3	174.6	355.1
54.3	99.9	129.8	284.0	72.7	144.5	138.1	355.3
54.9	101.5	132.4	288.8	59.9	101.5	194.1	355.5
55.5	103.3	135.2	293.9	127.5	103.3	125.0	355.8
56.1	105.1	138.1	299.3	60.8	162.6	132.4	355.9
56.8	107.1	141.1	305.0	66.3	109.1	180.5	355.9
57.5	109.1	144.4	311.1	61.8	149.8	144.4	356.0
58.3	111.3	147.9	317.5	65.0	155.8	135.2	356.0
59.1	113.6	151.5	324.3	74.8	121.6	159.7	356.1
59.9	116.1	155.5	331.5	77.1	131.5	147.9	356.5
60.8	118.8	159.7	339.3	59.1	179.9	118.6	357.5

continues on next page

Table 7 – continued from previous page

x_0	x_1	x_2	Sum	s_0	s_1	s_2	Max VaR
61.8	121.6	164.3	347.7	58.3	99.9	202.0	360.1
62.8	124.6	169.3	356.7	57.5	191.1	116.6	365.3
63.9	127.9	174.6	**366.4**	56.8	98.3	210.9	366.0
65.0	131.5	180.5	377.0	56.1	96.7	221.0	373.9
66.3	135.4	186.9	388.7	55.5	205.1	114.7	375.4
67.7	139.7	194.1	401.5	54.9	95.3	232.7	382.9
69.2	144.5	202.0	415.7	54.3	223.3	112.9	390.5
70.8	149.8	210.9	431.6	53.7	93.9	246.3	393.9
72.7	155.8	221.0	449.5	53.2	92.6	262.5	408.2
74.8	162.6	232.7	470.1	52.6	248.7	111.2	412.5
77.1	170.6	246.3	494.0	52.1	91.3	282.3	417.7
79.9	179.9	262.5	522.3	51.2	288.1	109.5	448.8
83.1	191.1	282.3	556.5	51.6	90.0	307.2	448.9
87.1	205.1	307.2	599.4	50.7	88.9	340.0	479.6
92.1	223.3	340.0	655.5	50.3	87.7	386.6	524.6
98.8	248.7	386.6	734.1	49.9	366.9	107.9	524.7
108.8	288.1	461.1	858.0	49.4	86.6	461.1	597.2
127.5	366.9	615.7	1110.1	49.0	85.6	615.7	750.3

The table illustrates the worst-case VaR may be substantially higher than when the marginals are perfectly correlated, here 45 percent higher at 352.8 vs. 242.5. The form of the output columns shows the two part structure. There is a series of values up to 356 involving moderate sized losses from each marginal with approximately the same total. The larger values of the rearrangement are formed from a large value from one marginal combined with smaller values from the other two.

The bold entry 366.4 indicates when the comonotonic sum of marginals exceeds the worst 0.99-VaR arrangement.

Performing the same calculation with $N = 1000$ samples from the largest 1 percent of each marginal produces an estimated worst VaR of 360.5.

The following code replicates this calculation in aggregate. The answer relies on random seeds and is slightly different from the table above.

```
In [1]: import aggregate as agg

In [2]: import numpy as np

In [3]: import pandas as pd

In [4]: import scipy.stats as ss

In [5]: ps = np.linspace(0.99, 1, 40, endpoint=False)

In [6]: params = {i: agg.mu_sigma_from_mean_cv(10, i) for i in [1,2,3]}

In [7]: df = pd.DataFrame({f'x_{i}': ss.lognorm(params[i][1],
   ...:     scale=np.exp(params[i][0])).isf(1-ps)
   ...:     for i in [1,2,3]}, index=ps)
   ...:

In [8]: df_ra = agg.rearrangement_algorithm_max_VaR(df)

In [9]: agg.qd(df_ra, float_format=lambda x: f'{x:.1f}', max_rows=100)

     x_1   x_2   x_3  total
30  75.3 120.1 157.6  352.9
11  69.7 117.4 166.5  353.6
25  68.2 151.4 134.3  353.8
```

(continues on next page)

(continued from previous page)

```
6   87.6 146.0 120.2  353.9
39  77.7 122.9 153.6  354.2
24  66.8 141.2 146.4  354.4
3   63.2 129.3 161.9  354.4
34  71.3 106.3 176.9  354.6
7   60.4 172.3 122.3  355.0
21  99.4 112.6 143.1  355.0
23  73.2 110.4 171.5  355.0
18 128.2 102.7 124.5  355.4
37  64.3 108.3 182.8  355.5
10  61.3 157.4 137.0  355.7
19 109.5 114.9 131.6  356.0
17  62.2 104.5 189.3  356.0
4   80.4 126.0 149.9  356.3
16  83.7 132.9 140.0  356.6
15  65.5 164.3 126.8  356.6
28  59.5 101.0 196.5  357.1
0   92.6 136.9 129.1  358.7
31  58.7 181.7 118.3  358.7
29  58.0  99.4 204.6  361.9
1   57.3 193.0 116.4  366.7
20  56.6  97.9 213.6  368.0
13  55.9  96.4 223.8  376.1
38  55.3 207.1 114.5  377.0
2   54.7  95.0 235.6  385.3
9   54.1 225.5 112.8  392.4
14  53.6  93.7 249.3  396.5
36  53.0  92.4 265.7  411.1
33  52.5 251.0 111.1  414.6
26  52.0  91.1 285.6  428.8
8   51.6 290.7 109.5  451.8
27  51.1  89.9 310.8  451.8
5   50.7  88.8 343.9  483.4
35  50.3 370.1 107.9  528.2
32  49.8  87.7 391.0  528.5
12  49.4  86.6 466.1  602.1
22  49.0  85.6 622.1  756.7
```

There are several important points to note about the Rearrangement Algorithm output and the failure of subadditivity it induces. They mirror the case $d = 2$.

- The dependence structure does not have right tail dependence.
- In Table 1, the comonotonic sum is greater than the maximum VaR sum for the top 40 percent observations, above 366.4. The algorithm output is tailored to a specific value of p and does not work for other ps. It produces relatively thinner tails for higher values of p than the comonotonic copula.
- The algorithm works for any non-trivial marginal distributions—it is universal.
- The implied dependence structure specifies only how the larger values of each marginal are related; any dependence structure can be used for values below VaR_p.

The Rearrangement Algorithm gives a definitive answer to the question "Just how bad could things get?" and perhaps provides a better base against which to measure diversification effect than either independence or the comonotonic copula. While the multivariate structure it reveals is odd and specific to p, it is not wholly improbable. It pinpoints a worst-case driven by a combination of moderately severe, but not extreme, tail event outcomes. Anyone who remembers watching their investment portfolio during a financial crisis has seen that behavior before! It is a valuable additional feature for any risk aggregation software.

CHAPTER

SIX

DESIGN AND DEVELOPMENT

6.1 Help Structure

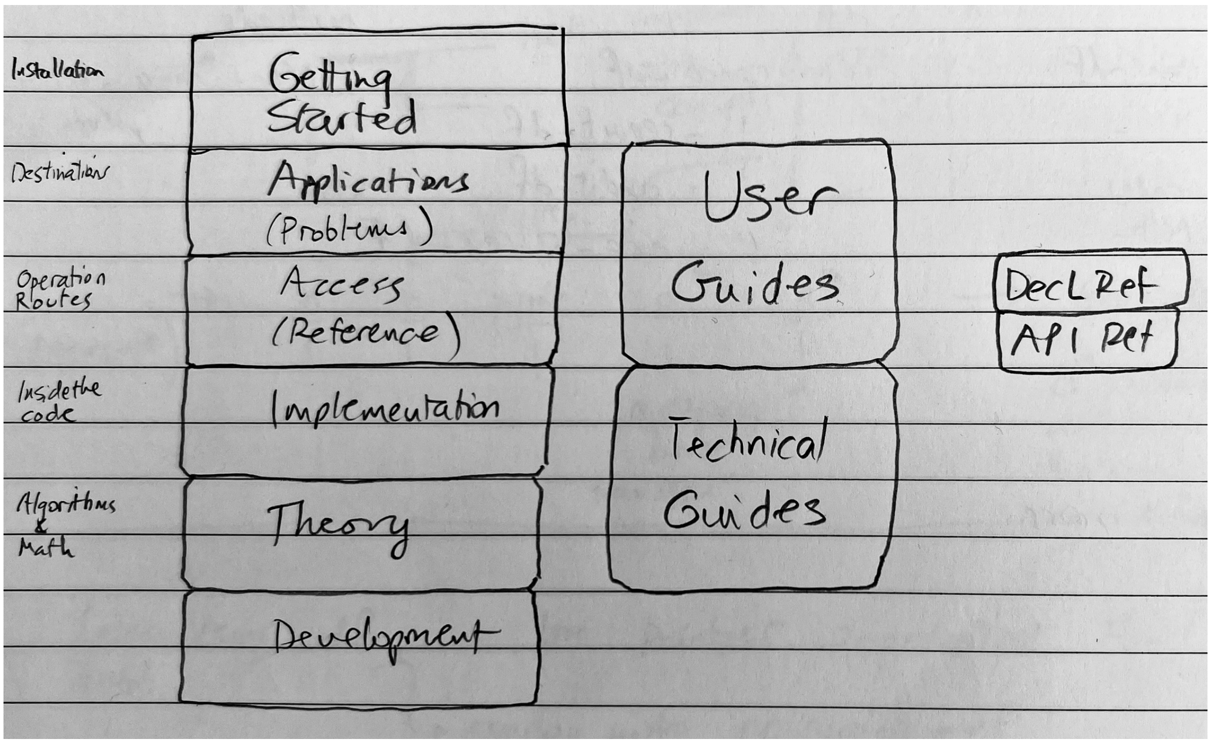


Fig. 1: Documentation roadmap.

The documentation is structured around application, access, implementation, and theory.

- **Application**: destinations, where you can go with the code and problems you can solve.
  - *User Guides*, problem sections
- **Access**: operating manual, how you access the functions you need to solve your problems
  - *User Guides*, problem sections
  - *API Reference*, traditional API reference documentation
  - *Dec Language Reference*
- **Implementation**: how the functionality is coded; algorithms.
  - *Technical Guides*
- **Theory**: the underlying mathematics.
  - *Technical Guides*

6.2 Design Philosophy

- Work at the correct speed order…
- …but don't worry about speed until it becomes a problem
- Save everything until space becomes an issue
- Offer sensible defaults for (almost) everything
- Separate internal naming from user naming and offer standard dataframe renamer dictionaries
- Use sensible number formats

6.3 History

I have built several iterations of software to work with aggregate distributions.

- Late 1990s: a Matlab tool for CNA Re with a graphical interface. Computed aggregates by business unit and the portfolio total. Used to discover the shocking fact there was only a 53 percent chance of achieving plan…which is obvious in hindsight but was a surprise at the time.
- Late 1990s: a C++ version of the Matlab code called SADCo in 1997-99. This code sits behind MALT. It won the CAS award for an online contribution.
- Early 2000s: another C++ version with an implementation of the Iman-Conover method to combine aggregates with correlation using the (gasp) normal copula, SCARE. Used by SCOR.
- 2003-2005: I worked on Aon Re Services' simulation based tools called the ALG (Aggregate Loss Generator) which simulated losses, and Prime/Re which manipulated the simulations and applied various reinsurance structures. ALG used a shared mixing variables approach to correlation.
- Late 2000s: At Aon Re, I also built related tools
  - The Levy measure maker
  - A simple approach to multi-year modeling based on re-scaling a base year, convolving using FFTs and tracking (and stopping) in default scenarios
- 2010s: At Aon Benfield, I was involved with ReMetrica, a very sophisticated, general purpose DFA/ERM simulation tool.

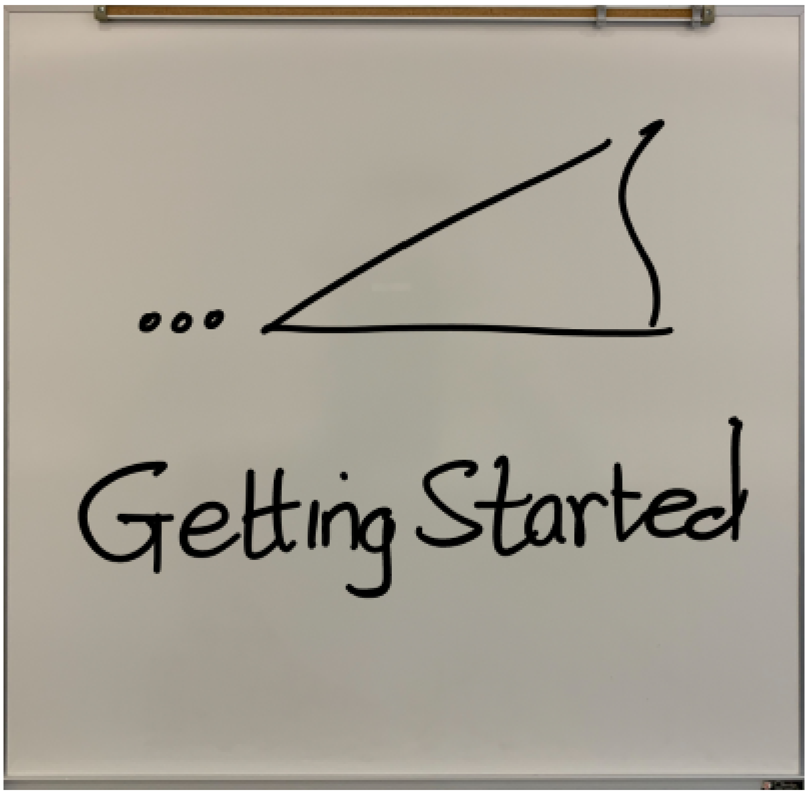
CHAPTER

SEVEN

INTRODUCTION

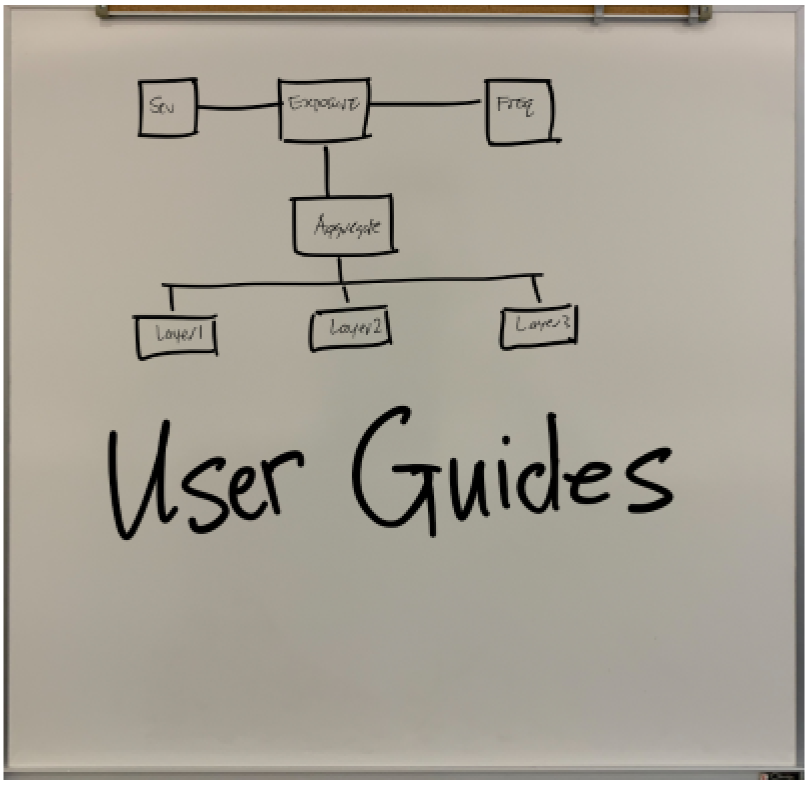
`aggregate` builds approximations to compound (aggregate) probability distributions quickly and accurately. It can be used to solve insurance, risk management, and actuarial problems using realistic models that reflect underlying frequency and severity. It delivers the speed and accuracy of parametric distributions to situations that usually require simulation, making it as easy to work with an aggregate (compound) probability distribution as the lognormal. `aggregate` includes an expressive language called DecL to describe aggregate distributions and is implemented in Python under an open source BSD-license.

This help document is in six parts plus a bibliography.



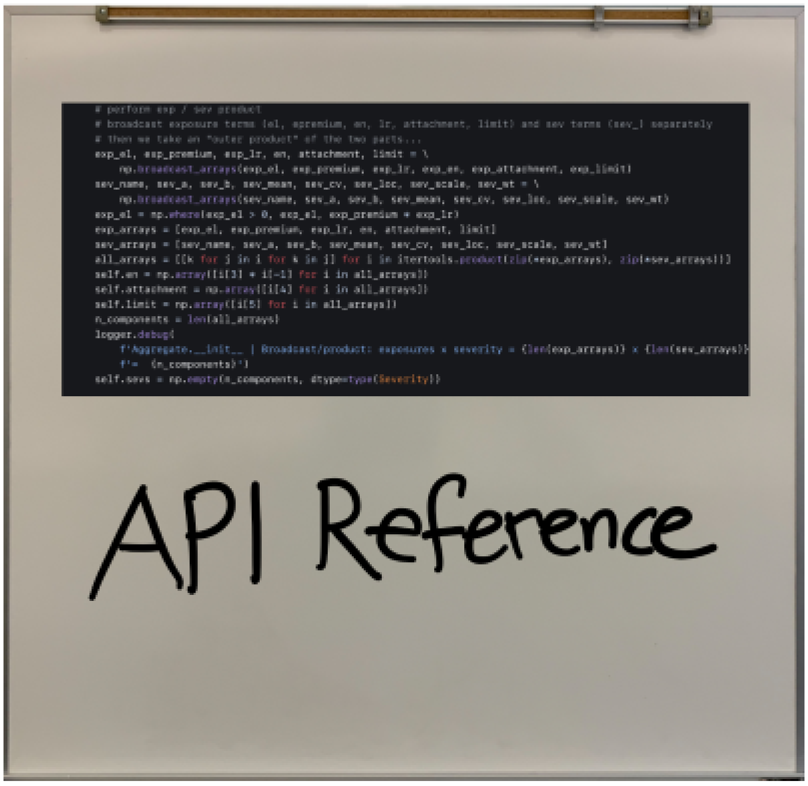
Get up and running: installation, `aggregate` "hello world", and a glimpse into the functionality.

Getting Started



How to solve real-world actuarial problems using `aggregate`.

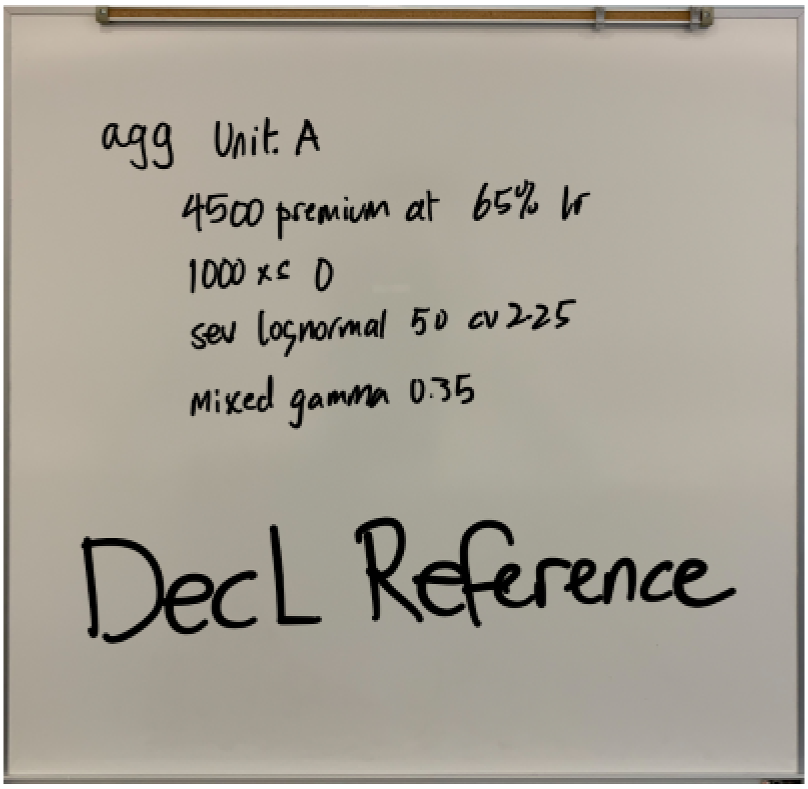
User Guides



```
# perform exp / sev product
# broadcast exposure terms (el, epremium, en, lr, attachment, limit) and sev terms (sev_) separately
# then we take an "outer product" of the two parts...
exp_el, exp_premium, exp_lr, en, attachment, limit = \
    np.broadcast_arrays(exp_el, exp_premium, exp_lr, exp_en, exp_attachment, exp_limit)
sev_name, sev_a, sev_b, sev_mean, sev_cv, sev_loc, sev_scale, sev_wt = \
    np.broadcast_arrays(sev_name, sev_a, sev_b, sev_mean, sev_cv, sev_loc, sev_scale, sev_wt)
exp_el = np.where(exp_el > 0, exp_el, exp_premium * exp_lr)
exp_arrays = [exp_el, exp_premium, exp_lr, en, attachment, limit]
sev_arrays = [sev_name, sev_a, sev_b, sev_mean, sev_cv, sev_loc, sev_scale, sev_wt]
all_arrays = [[k for j in i for k in j] for i in itertools.product(zip(*exp_arrays), zip(*sev_arrays))]
self.en = np.array([i[3] * i[-1] for i in all_arrays])
self.attachment = np.array([i[4] for i in all_arrays])
self.limit = np.array([i[5] for i in all_arrays])
n_components = len(all_arrays)
logger.debug(
    f'Aggregate.__init__ | Broadcast/product: exposures x severity = {len(exp_arrays)} x {len(sev_arrays)}
    f'=  {n_components}')
self.sevs = np.empty(n_components, dtype=type(Severity))
```

Documentation for every class and function, for developers and more advanced users.

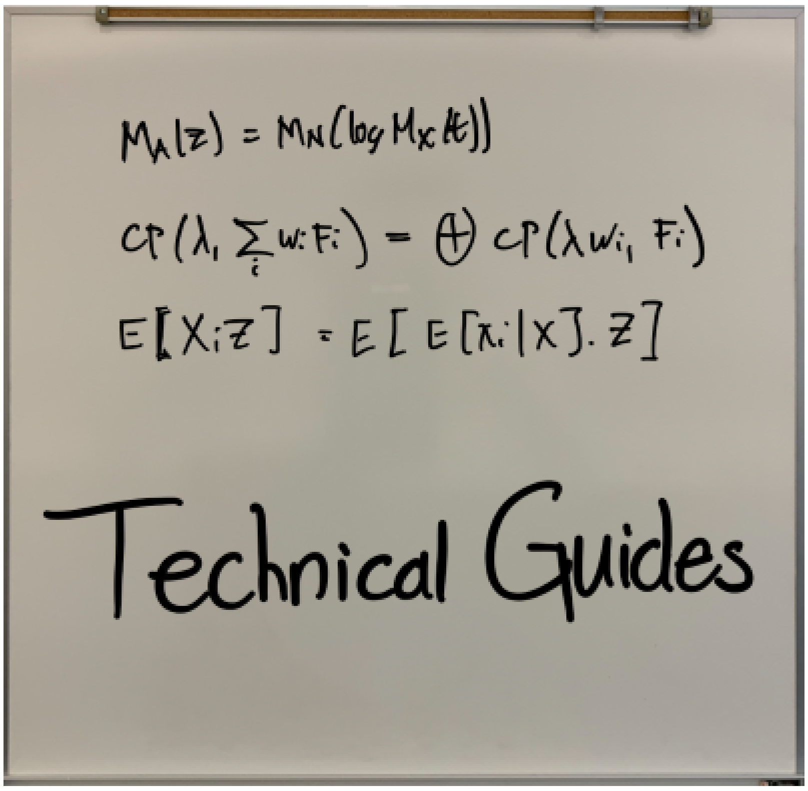
API Reference



```
agg Unit.A
    4500 premium at 65% lr
    1000 xs 0
    sev lognormal 50 cv 2.25
    mixed gamma 0.35
```

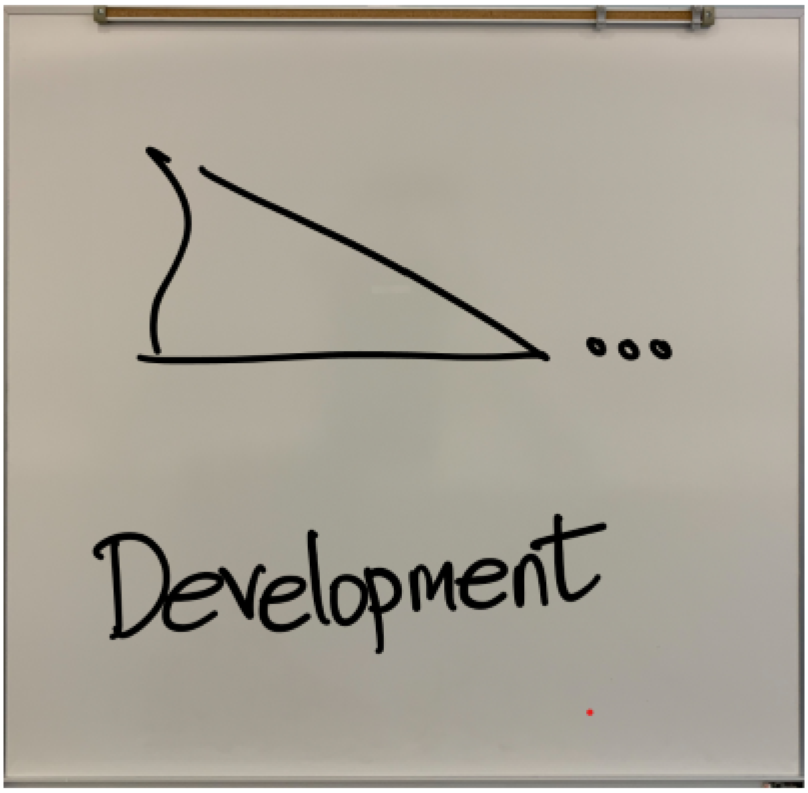
The Dec Language (DecL) for specifying aggregate distributions.

DecL Reference


$M_A(z) = M_N(\log M_X(t))$
$CP(\lambda, \sum_i w_i F_i) = \bigoplus CP(\lambda w_i, F_i)$
$E[X_i z] = E[E[X_i \mid X] . z]$
Technical Guides

Probability theory background and the numerical implementation methods employed by `aggregate`.

Technical Guides



Design philosophy, competing products, future development ideas, and historical perspective.

Development

BIBLIOGRAPHY

[Ace02] Carlo Acerbi. Spectral measures of risk: A coherent representation of subjective risk aversion. *Journal of Banking & Finance*, 26(7):1505–1518, jul 2002. URL: http://linkinghub.elsevier.com/retrieve/pii/S0378426602002819, doi:10.1016/S0378-4266(02)00281-9.

[ABT17] Hansjörg Albrecher, Jan Beirlant, and Jozef L Teugels. *Reinsurance: Actuarial and Statistical Aspects*. John Wiley & Sons, 2017.

[AD88] RR Anderson and Wemin Dong. Pricing catastrophe reinsurance with reinstatement provisions using a catastrophe model. *Casualty Actuarial Society Forum*, pages 303–322, 1988.

[AGK19] Richard Arratia, Larry Goldstein, and Fred Kochman. Size bias for one and all. *Probability Surveys*, 16:1–61, 2019. arXiv:1308.2729, doi:10.1214/13-PS221.

[Bah15] David Bahnemann. *Distributions for Actuaries*. Casualty Actuarial Society Mongraphs No. 2, 2015. ISBN 9780962476280. URL: www.casact.org.

[BN90] R.A. Bear and K.J. Nemlick. Pricing the impact of adjustable features and loss sharing provisions of reinsurance treaties. *Proceedings of the Casualty Actuarial Society*, 77(147):86–87, 1990. doi:10.1016/0167-6687(93)91078-9.

[Ber97] Stefan Bernegger. The Swiss Re exposure curves and the MBBEFD distribution class. *ASTIN Bulletin*, 27(1):99–111, 1997.

[Ber83] J Bertram. Calculations of aggregate claims distributions in case of negative risk sums. In *17th ASTIN Colloquium, Lindau, Germany*. 1983.

[Bil86] Patrick Billingsley. *Probability and Measure*. J. Wiley and Sons, second edition, 1986.

[Bod07] Neil M. Bodoff. Capital Allocation by Percentile Layer. *Variance*, 3(1):13–30, 2007.

[BTWuthrich17] Tim J. Boonen, Andreas Tsanakas, and Mario V. Wüthrich. Capital allocation for portfolios with non-linear risk aggregation. *Insurance: Mathematics and Economics*, 72:95–106, 2017. doi:10.1016/j.insmatheco.2016.11.003.

[BV10] Jonathan M Borwein and Jon D Vanderwerff. *Convex Functions - Construction, Characterizations and Counterexamples*. Cambridge University Press, 2010. ISBN 9780521850056.

[BGH+97] Newton Bowers, Hans Gerber, James Hickman, Donald Jones, and Cecil Nesbitt. *Actuarial Mathematics*. Society of Actuaries, 1997. doi:10.2307/253313.

[BPVS07] Paul J Brehm, Geoffrey R Perry, Gary G Venter, and E Witcraft Susan. Enterprise risk analysis for property & liability insurance companies. *Guy Carpenter & Company, LLC*, 2007.

[But94] Robert P Butsic. Solvency Measurement for Property-Liability Risk-Based Capital Applications. *The Journal of Risk and Insurance*, 61(4):656–690, 1994. URL: http://www.jstor.org/stable/253643%5Cnhttp://www.jstor.org/page/.

[Buhlmann84] Hans Bühlmann. Numerical evaluation of the compound Poisson distribution: Recursion or fast fourier transform? *Scandinavian Actuarial Journal*, 1984(2):116–126, 1984. doi:10.1080/03461238.1984.10413759.

[CJP13] Luciano Campi, Elyès Jouini, and Vincent Porte. Efficient portfolios in financial markets with proportional transaction costs. *Mathematics and Financial Economics*, 7(3):281–304, 2013. doi:10.1007/s11579-013-0099-4.

[CD03] G. Carlier and R. A. Dana. Core of convex distortions of a probability. *Journal of Economic Theory*, 113(2):199–222, 2003. doi:10.1016/S0022-0531(03)00122-4.

[CM99] Peter Carr and Dilip Madan. Option valuation using the fast Fourier transform. *The Journal of Computational Finance*, 2(4):61–73, 1999. doi:10.21314/jcf.1999.043.

[Car13] Robert L Carter. *Reinsurance*. Springer Science & Business Media, 2013.

[CO11] Alexander Cherny and Dmitri Orlov. On two approaches to coherent risk contribution. *Mathematical Finance*, 21(3):557–571, 2011. doi:10.1111/j.1467-9965.2010.00441.x.

[Cla14] David R Clark. Basics of Reinsurance Pricing Actuarial Study Note. *CAS Study Note*, 2014. URL: http://www.casact.org/library/studynotes/Clark_2014.pdf.

[CT04] David R Clark and Charles A Thayer. A primer on the exponential family of distributions. *Casualty Actuarial Society Spring Forum*, pages 117–148, 2004. URL: papers2://publication/uuid/C7FF0B8F-F767-4F09-9714-F0D3711A30D8.

[Con99] W J Conover. *Practical nonparametric statistics*. John Wiley and Sons, third edition, 1999.

[CS73] P. C. Consul and L. R. Shenton. Some interesting properties of Lagrangian distributions. *Communications in Statistics*, 2(3):263–272, 1973. doi:10.1080/03610927308827073.

[CODonnell09] Christopher L. Culp and Kevin J. O'Donnell. Catastrophe reinsurance and risk capital in the wake of the credit crisis. *The Journal of Risk Finance*, 10(5):430–459, 2009. URL: http://www.emeraldinsight.com/10.1108/15265940911001367, doi:10.1108/15265940911001367.

[CP05] J. David Cummins and Richard D. Phillips. Estimating the Cost of Equity Capital for Property-Liability Insurers. *Journal of Risk and Insurance*, 72(3):441–478, 2005. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1539-6975.2005.00132.x/full.

[DPP93] Chris D Daykin, Teivo Pentikainen, and Martti Pesonen. *Practical risk theory for actuaries*. Chapman and Hall/CRC, 1993.

[Del00] Freddy Delbaen. Coherent risk measures (Pisa Notes). *Pisa Notes*, 24(4):733–739, 2000. doi:10.1007/BF02809088.

[Den01] Michel Denault. Coherent allocation of risk capital. *The Journal of Risk*, 4(1):1–34, 2001. URL: ftp://ftp.sam.math.ethz.ch/pub/risklab/papers/CoherentAllocation.pdf, doi:10.21314/jor.2001.053.

[Den19] Michel Denuit. Size-biased transform and conditional mean risk sharing, with application to p2p insurance and tontines. *ASTIN Bulletin*, 49(3):591–617, 2019. doi:10.1017/asb.2019.24.

[DD12] Michel Denuit and Jan Dhaene. Convex order and comonotonic conditional mean risk sharing. *Insurance: Mathematics and Economics*, 51(2):265–270, 2012. URL: http://dx.doi.org/10.1016/j.insmatheco.2012.04.005, doi:10.1016/j.insmatheco.2012.04.005.

[DHR22] Michel Denuit, Peter Hieber, and Christian Y. Robert. Mortality Credits Within Large Survivor Funds. *ASTIN Bulletin*, 52(3):813–834, 2022. doi:10.1017/asb.2022.13.

[DR20] Michel M. Denuit and C Y Robert. Risk Reduction by Conditional Mean Risk Sharing With Application to Collaborative Insurance. Technical Report, UC Louvain, 2020.

[DKLT12] Jan Dhaene, Alexander Kukush, Daniel Linders, and Qihe Tang. Remarks on quantiles and distortion risk measures. *European Actuarial Journal*, 2(2):319–328, 2012.

[DHardleG12] Jin Chuan Duan, Wolfgang Karl Härdle, and James E. Gentle. Handbook of computational finance. *Handbook of Computational Finance*, pages 1–804, 2012. doi:10.1007/978-3-642-17254-0.

[EGrubelP93] P. Embrechts, R. Grübel, and S. M. Pitts. Some applications of the fast Fourier transform algorithm in insurance mathematics. *Statistica Neerlandica*, 47(1):59–75, 1993. doi:10.1111/j.1467-9574.1993.tb01406.x.

[EF09] Paul Embrechts and Marco Frei. Panjer recursion versus FFT for compound distributions. *Mathematical Methods of Operations Research*, 69(3):497–508, 2009. doi:10.1007/s00186-008-0249-2.

[EKluppelbergM97] Paul Embrechts, Claudia Klüppelberg, and Thomas Mikosch. *Modelling Extremal Events*. Springer Verlag, Berlin Heidelberg, 1997. doi:10.1007/978-3-642-33483-2.

[EPR13] Paul Embrechts, Giovanni Puccetti, and Ludger Ruschendorf. Model uncertainty and VaR aggregation. *Journal of Banking and Finance*, 37(8):2750–2764, 2013. doi:10.1016/j.jbankfin.2013.03.014.

[Fel71] William Feller. *An Introduction to Probability Theory and its Applications, Volume 2*. J. Wiley and Sons, second edition, 1971. ISBN 0471257095.

[FMPP19] Ginda Fisher, Lawrence McTaggart, Jill Petker, and Rebecca Pettingell. Individual Risk Rating Study Note. *CAS Exam 8 Study Note*, 2019.

[FollmerS11] Hans Föllmer and Alexander Schied. *Stochastic finance: an introduction in discrete time*. Walter de Gruyter, third edit edition, 2011.

[FollmerS16] Hans Föllmer and Alexander Schied. *Stochastic Finance: An Introduction in Discrete Time*. Walter de Gruyter, Berlin, Boston, fourth edition, 2016. ISBN 9788578110796. arXiv:arXiv:1011.1669v3, doi:10.1017/CBO9781107415324.004.

[Ger82] Hans U. Gerber. On the numerical evaluation of the distribution of aggregate claims and its stop-loss premiums. *Insurance Mathematics and Economics*, 1(1):13–18, 1982. doi:10.1016/0167-6687(82)90016-6.

[Gra97] Jan Grandell. *Mixed poisson processes*. Volume 77. CRC Press, 1997.

[GS07] Helmut Grundl and Hato Schmeiser. Capital allocation for insurance companies—What Good Is It? *Journal of Risk and Insurance*, 2007. URL: http://www.jstor.org/stable/2691539.

[GrubelH99] Rudolf Grübel and Renate Hermesmeier. Computation of Compound Distributions I: Aliasing Errors and Exponential Tilting. *Astin Bulletin*, 29(2):197–214, 1999. doi:10.2143/AST.29.2.504611.

[GrubelH00] Rudolf Grübel and Renate Hermesmeier. Computation of Compound Distributions II: Discretization Errors and Richardson Extrapolation. *ASTIN Bulletin*, 30(2):309–332, 2000. doi:10.2143/AST.30.2.504638.

[HM83] Philip E Heckman and Glenn G Meyers. The calculation of aggregate loss distributions from claim severity and claim count distributions. *Proceedings of the Casualty Actuarial Society*, pages 49–66, 1983.

[HC03] David L Homer and David R Clark. Insurance Applications of Bivariate Distributions. *Proceedings of the Casualty Actuarial Society*, 90(iid):274–307, 2003. URL: http://www.casact.org/pubs/proceed/proceed03/03274.pdf.

[HF96] Rob J. Hyndman and Yanan Fan. Sample Quantiles in Statistical Packages. *American Statistician*, 50(4):361–365, 1996. doi:10.1080/00031305.1996.10473566.

[Hurlimann86] W. Hürlimann. Error Bounds for Stop-loss Premiums Calculated with the Fast Fourier Transform. *Scandinavian Actuarial Journal*, 1986(2):107–113, 1986. doi:10.1080/03461238.1986.10413798.

[IJW10] Rustam Ibragimov, Dwight Jaffee, and Johan Walden. Pricing and Capital Allocation for Multiline Insurance Firms. *Journal of Risk and Insurance*, 77(3):551–578, mar 2010. URL: http://doi.wiley.com/10.1111/j.1539-6975.2010.01353.x, doi:10.1111/j.1539-6975.2010.01353.x.

[Jew22a] Stephen Jewson. Application of uncertain hurricane climate change projections to catastrophe risk models. *Stochastic Environmental Research and Risk Assessment*, 2022. URL: https://doi.org/10.1007/s00477-022-02198-y, doi:10.1007/s00477-022-02198-y.

[Jew22b] Stephen Jewson. Projections of Changes in U.S. Hurricane Damage Due to Projected Changes in Hurricane Frequencies. *submitted*, 2022.

[JKK05] Norman L Johnson, Samuel Kotz, and Adrienne W Kemp. *Univariate discrete distributions*. John Wiley & Sons, 2005.

[JK01] Elyès Jouini and Hédi Kallal. Efficient trading strategies in the presence of market frictions. *Review of Financial Studies*, 14(2):343–369, 2001. doi:10.1093/rfs/14.2.343.

[Jorgensen97] Bent Jørgensen. *The theory of dispersion models*. CRC Press, 1997.

[KGDD08] Rob Kaas, Marc Goovaerts, Jan Dhaene, and Michel Denuit. *Modern Actuarial Risk Theory*. Springer, 2008. ISBN 978-3-540-70992-3. arXiv:arXiv:1011.1669v3, doi:10.1007/978-3-540-70998-5.

[KPW19] Stuart A Klugman, Harry H Panjer, and Gordon E Willmot. *Loss Models: From Data to Decisions*. Volume 715. John Wiley & Sons, 5 edition, 2019.

[Kus01] Shigeo Kusuoka. On law invariant coherent risk measures. *Advances in Mathematical Economics*, 3:83–95, 2001. URL: http://link.springer.com/chapter/10.1007/978-4-431-67891-5_4.

[Korner22] Thomas William Körner. *Fourier analysis*. Cambridge university press, 2022.

[Loe55] Michel Loeve. *Probability Theory*. D. Van Nostrand Company, 1955. ISBN 9780486814889. doi:10.1137/1006078.

[Loe17] Michel Loeve. *Probability theory*. Courier Dover Publications, 2017.

[Lud91] Stephen J Ludwig. An Exposure Rating Approach to Pricing Excess-Of-Loss Reinsurance. *Proceedings of the Casualty Actuarial Society*, 1991.

[Luk70] Eugene Lukacs. *Characteristic Functions*. Griffin, London, 2 edition, 1970.

[LS09] Xiaolin Luo and Pavel V Shevchenko. Computing Tails of Compound Distributions Using Direct Numerical Integration. *Journal of Computational Finance*, 13(2):1–33, 2009.

[LS11] Xiaolin Luo and Pavel V Shevchenko. A Short Tale of Long Tail Integration. *Numerical Algorithms*, 56(4):577–590, 2011. arXiv:arXiv:1005.1705v1.

[Maj18] John A. Major. Distortion Measures on Homogeneous Financial Derivatives. *Insurance: Mathematics and Economics*, 79:82–91, 2018. URL: http://www.ssrn.com/abstract=2972955http://linkinghub.elsevier.com/retrieve/pii/S0167668717303384, doi:10.2139/ssrn.2972955.

[MM20] John A. Major and Stephen J. Mildenhall. Pricing and Capital Allocation for Multiline Insurance Firms With Finite Assets in an Imperfect Market. *Arxiv*, pages 1–33, 2020. URL: http://arxiv.org/abs/2008.12427, arXiv:2008.12427.

[MAKL95] Paul Malliavin, Hélène Airault, Leslie Kay, and Gérard Letac. *Integration and probability*. Volume 157. Springer Science & Business Media, 1995.

[Man05] Donald Mango. Insurance Capital as a Shared Asset. *Astin Bulletin*, 35(2):471–486, 2005. URL: https://www.beanactuary.com/pubs/forum/06fforum/577.pdf.

[MMAB13] Donald Mango, John Major, Avraham Adler, and Claude Bunick. Capital Tranching: A RAROC Approach to Assessing Reinsurance Cost Effectiveness. *Variance*, 7(September):82–91, 2013. URL: http://www.actuaries.org.uk/sites/all/files/documents/pdf/capital-tranching-raroc-approach-assessing-reinsurance-cost-effectiveness.pdf.

[MM04] Massimo Marinacci and Luigi Montrucchio. A characterization of the core of convex games through Gateaux derivatives. *Journal of Economic Theory*, 116(2):229–248, 2004. URL: http://dx.doi.org/10.1016/S0022-0531(03)00258-8, doi:10.1016/S0022-0531(03)00258-8.

[MPFV02] Ana J Mata, D Ph, Brian Fannin, and Mark A Verheyen. Pricing Excess of Loss Treaty with Loss Sensitive Features: An Exposure Rating Approach. In *General Insurance Convention*. 2002.

[MN19] Peter McCullagh and John A Nelder. *Generalized linear models*. Routledge, 2019.

[McG69] John S McGuinness. Is “probable maximum loss” (PML) a useful concept? *Proceedings of Casualty Actuarial Society*, LVI(May):31–48, 1969.

[McK14] Henry McKean. *Probability: the classical limit theorems*. Cambridge University Press, 2014.

[MR06] Christian Menn and Svetlozar T Rachev. Calibrated FFT-based density approximations for α-stable distributions. *Computational Statistics and Data Analysis*, 50(8):1891–1904, 2006. doi:10.1016/j.csda.2005.03.004.

[Mey96] Glenn G Meyers. The competitive market equilibrium risk load formula for catastrophe ratemaking. *PCAS*, pages 563–600, 1996.

[Mey19] Glenn G Meyers. A Cost of Capital Risk Margin Formula For Non-Life Insurance Liabilities. *Variance*, 12(2):186–198, 2019.

[Mil04] Stephen J Mildenhall. A Note on the Myers and Read Capital Allocation Formula. *North American Actuarial Journal*, 8(2):32–44, 2004. URL: http://library.soa.org/library-pdf/naaj0402_3.pdf.

[Mil05] Stephen J Mildenhall. Correlation and Aggregate Loss Distributions With An Emphasis On The Iman-Conover Method. *Casualty Actuarial Society Forum*, 2005.

[Mil17] Stephen J. Mildenhall. Actuarial Geometry. *Risks*, 2017. doi:10.3390/risks5020031.

[Mil22] Stephen J. Mildenhall. Similar Risks Have Similar Prices: A Useful and Exact Quantification. *Insurance: Mathematics and Economics*, 105:203–210, 2022. URL: https://doi.org/10.1016/j.insmatheco.2022.04.006, doi:10.1016/j.insmatheco.2022.04.006.

[MM22a] Stephen J. Mildenhall and John A. Major. *Pricing Insurance Risk: Theory and Practice*. John Wiley & Sons, Inc., 2022.

[MM22b] Stephen J. Mildenhall and John A. Major. *Pricing Insurance Risk: Theory and Practice*. John Wiley & Sons, Inc., 2022. ISBN 9781119130536.

[MP98] Moshe Arye Milevsky and Steven E. Posner. Asian Options, the Sum of Lognormals , and the Reciprocal Gamma Distribution. *Journal of Financial and Quantitative Analysis*, 33(3):409–422, 1998.

[MWJHF17] Kirsten Mitchell-Wallace, Matthew Jones, John Hillier, and Matthew Foote. *Natural Catastrophe Risk Managment and Modeling - A Practitioner's Guide*. Wiley-Blackwell, 2017.

[MC87] Stewart C Myers and Richard A Cohn. A discounted cash flow approach to property-liability insurance rate regulation. In *Fair Rate of Return in Property-Liability Insurance*, pages 55–78. Springer, 1987.

[MReadJr01] Stewart C Myers and James A Read Jr. Capital allocation for insurance companies. *Journal of Risk and Insurance*, 68(4):545–580, 2001. URL: http://www.jstor.org/stable/2691539.

[PW92] Harry H Panjer and Gordon E Willmot. *Insurance risk models*. Society of Actuaries, 1992.

[PL83] Harry H. Panjer and B. W. Lutek. Practical aspects of stop-loss calculations. *Insurance: Mathematics and Economics*, 2:159–177, 1983.

[PPP01] Dmitry E Papush, Gary S Patrik, and Felix Podgaits. Approximations of the Aggregate Loss Distribution. *Casualty Actuarial Society Forum*, Winter:175–186, 2001. URL: http://www.casact.org/pubs/forum/01wforum/01wf175.pdf.

[PPZ21] Dmitry E Papush, Aleksey S Popelyukhin, and Jasmine G Zhang. Approximating the Aggregate Loss Distribution. *Variance*, 14(2):1–10, 2021.

[Par15] Pietro Parodi. *Pricing in General Insurance*. CRC Press, 2015. ISBN 9781466581487.

[PCA98] Richard D. Phillips, J. David Cummins, and Franklin Allen. Financial Pricing of Insurance in the Multiple-Line Insurance Company. *Journal of Risk and Insurance*, 65(4):597–636, 1998. URL: http://www.jstor.org/stable/253804http://www.jstor.org/stable/10.2307/253804http://www.jstor.org/stable/253804%5Cnhttp://www.jstor.org/stable/10.2307/253804, doi:10.2307/253804.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd editio edition, 1992. ISBN 0521431085.

[PR12] Giovanni Puccetti and Ludger Ruschendorf. Computation of sharp bounds on the distribution of a function of dependent risks. *Journal of Computational and Applied Mathematics*, 236(7):1833–1840, 2012. URL: http://dx.doi.org/10.1016/j.cam.2011.10.015, doi:10.1016/j.cam.2011.10.015.

[Rob92] John P. Robertson. The computation of aggregate loss distributions. *Proceedings of the Casualty Actuarial Society*, 79(150):57–133, 1992.

[RR14] R. T. Rockafellar and J. O. Royset. Random variables, monotone relations, and convex analysis. *Mathematical Programming*, 148(1-2):297–331, 2014. doi:10.1007/s10107-014-0801-1.

[SW14] Adrien Saumard and Jon A. Wellner. Log-concavity and strong log-concavity: a review. *Statistics Surveys*, 8:45–114, 2014. arXiv:1404.5886, doi:10.1214/14-SS107.

[ST08] P Schaller and G Temnov. Efficient and Precise Computation of Convolutions: Applying FFTs to Heavy Tailed Distributions. *Computational Methods in Applied Mathematics*, 8(2):187–200, 2008.

[SDRuszczynski09] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczyński. *Lectures on Stochastic Programming*. Number May. Society for Industrial and Applied Mathematics, 2009. ISBN 978-0-89871-687-0. URL: http://epubs.siam.org/doi/book/10.1137/1.9780898718751, arXiv:arXiv:1011.1669v3, doi:10.1137/1.9780898718751.

[She06] Michael Sherris. Solvency, capital allocation, and fair rate of return in insurance. *Journal of Risk and Insurance*, 73(1):71–96, 2006. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-2966.2006.00166.x/full.

[She10] Pavel V. Shevchenko. Calculation of aggregate loss distributions. *Journal of Operational Risk*, 5(2):3–40, 2010. URL: http://arxiv.org/abs/1008.1108, arXiv:1008.1108.

[SS11] Elias M Stein and Rami Shakarchi. *Fourier analysis: an introduction*. Volume 1. Princeton University Press, 2011.

[SW71] Elias M Stein and Guido Weiss. *Introduction to Fourier analysis on Euclidean spaces*. Volume 1. Princeton university press, 1971.

[Str97] Robert W. Strain. *Reinsurance*. Robert W. Strain Publishing & Seminars, Incorporated, 1997.

[Str86] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[Svi10] Gregor Svindland. Continuity properties of law-invariant (quasi-)convex risk functions on $L^\infty $. *Mathematics and Financial Economics*, 3(1):39–43, 2010. doi:10.1007/s11579-010-0026-x.

[Tas99] Dirk Tasche. Risk contributions and performance measurement. *Report of the Lehrstuhl fur mathematische Statistik, TU Munchen*, pages 1–26, 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9393&rep=rep1&type=pdf.

[TW08] Grigory Temnov and Richard Warnung. A Comparison of Loss Aggregation Methods for Operational Risk. *Journal of Operational Risk*, 3(1):1–22, 2008.

[Ter13] Audrey Terras. *Harmonic analysis on symmetric spaces—Euclidean Space, the Sphere, and the Poincar\/e Upper Half-Plane*. Springer Science & Business Media, 2013.

[TB03] Andreas Tsanakas and Christopher Barnett. Risk capital allocation and cooperative pricing of insurance liabilities. *Insurance: Mathematics and Economics*, 33(2):239–254, 2003. doi:10.1016/S0167-6687(03)00137-9.

[VMK06] Gary G. Venter, John A. Major, and Rodney E. Kreps. Marginal Decomposition of Risk Measures. *ASTIN Bulletin*, 36(2):375–413, oct 2006. URL: http://poj.peeters-leuven.be/content.php?url=article&id=2017927, doi:10.2143/AST.36.2.2017927.

[Ver04] R J Verrall. Bayesian Generalized Linear Model for the Bornhuetter-Furguson Method of Claims Reserving. *North American Actuarial Journal*, 8(3):67–89, 2004.

[Vit90] Richard A. Vitale. On stochastic dependence and a class of degenerate distributions. *Lecture Notes-Monograph Series*, pages 459–469, 1990. URL: http://projecteuclid.org/euclid.lnms/1215457581.

[Wan95] Shaun S. Wang. Insurance pricing and increased limits ratemaking by proportional hazards transforms. *Insurance: Mathematics and Economics*, 17(1):43–54, 1995. URL: http://dx.doi.org/10.1016/0167-6687(95)00010-P, doi:10.1016/0167-6687(95)00010-P.

[Wan96] Shaun S. Wang. Premium Calculation by Transforming the Layer Premium Density. *ASTIN Bulletin*, 26(01):71–92, 1996. URL: https://www.cambridge.org/core/product/identifier/S0515036100003214/type/journal_article, doi:10.2143/AST.26.1.563234.

[Wan98] Shaun S. Wang. Aggregation of correlated risk portfolios: models and algorithms. *Proceedings of the Casualty Actuarial society*, pages 848–939, 1998. URL: http://www.casact.com/pubs/proceed/proceed98/980848.pdf.

[Wan00] Shaun S. Wang. A Class of Distortion Operators for Pricing Financial and Insurance Risks. *The Journal of Risk and Insurance*, 67(1):15–36, 2000. doi:10.2307/253675.

[Wan02] Shaun S. Wang. A Universal Framework for Pricing Financial and Insurance Risks. *ASTIN Bulletin*, 32(2):213–234, 2002. doi:10.2143/AST.32.2.1027.

[Wil91] David Williams. *Probability with martingales*. Cambridge university press, 1991.

[WK16] Huon Wilson and Uri Keich. Accurate pairwise convolutions of non-negative vectors via FFT. *Computational Statistics and Data Analysis*, 101:300–315, 2016. URL: http://dx.doi.org/10.1016/j.csda.2016.03.010, doi:10.1016/j.csda.2016.03.010.

[Woo02] G. Woo. Natural Catastrophe Probable Maximum Loss. *British Actuarial Journal*, 8(5):943–959, 2002. doi:10.1017/s1357321700004037.

[vCerny04] Aleš Černý. Introduction to Fast Fourier Transform in Finance. *Journal of Derivatives*, 12(1):73–88, 2004.

[AonBenfield15] Aon Benfield. Insurance Risk Study, Ed. 10. Technical Report, Aon Limited, 2015.

[DeWaegenaereKL03] Anja De Waegenaere, Robert Kast, and Andre Lapied. Choquet pricing and equilibrium. *Insurance: Mathematics and Economics*, 32:359–370, 2003. doi:10.1016/S0167-6687(03)00116-1.

PYTHON MODULE INDEX

INDEX