Aggregate: Fast, accurate, and flexible approximation of compound probability distributions

Release 1.0

Stephen J. Mildenhall

Jul 23, 2023

Contents

1 Abstract

`Aggregate` implements an efficient FFT-based algorithm to approximate compound probability distributions. Leveraging FFT-based methods offers advantages over recursion and simulation-based approaches, providing speed and accuracy to otherwise time-consuming calculations. Combining user-friendly features and an expressive domain-specific language called DecL, `Aggregate` enables practitioners and non-programmers to work with complex distributions effortlessly. The software verifies the accuracy of its FFT-based numerical approximations by comparing their first three moments to those calculated analytically from the specified frequency and severity. This moment-based validation, combined with carefully chosen default parameters, allows users without in-depth knowledge of the underlying algorithm to be confident in the results. `Aggregate` supports a wide range of frequency and severity distributions, policy limits and deductibles, and reinsurance structures and has applications in pricing, reserving, risk management, teaching, and research. It is written in Python.

Keywords: aggregate distribution, compound distribution, frequency, severity, insurance, collective risk model, Fourier transform, FFT, characteristic functions, reinsurance, risk management, Python, risk aggregation

JEL Codes: C52, C63, C65, C88, G22, G32

2 Introduction

2.1 Background and motivation

Frequency-severity compound distributions form the basis of the collective risk model, a fundamental concept in actuarial science. Compound distributions have widespread applications in actuarial pricing, reserving, risk management, economic capital modeling, and solvency assessment [Albrecher *et al.*, 2017, Bahnemann, 2015, Frees, 2018, Klugman *et al.*, 2019, Parodi, 2015]. They are crucial in computing various statistics such as point estimates, higher moments, reasonable ranges, quantiles, VaR, and TVaRs. They are integral to classical premium calculation principles and risk measures [Goovaerts *et al.*, 1984, Mildenhall and Major, 2022]. Despite their importance in actuarial practice, compound distributions pose significant computational challenges due to their lack of a closed-form expression.

There are two challenges to address when computing a compound: how to **specify** it flexibly yet succinctly and how to **estimate** its distribution probabilities quickly and accurately.

Specifying real-world compound distributions is inherently complex: they can incorporate multiple mixed severity curves, varying policy limits and attachments, shared frequency mixing distributions, and multi-layer reinsurance structures. `Aggregate` addresses this challenge through its first of three key innovations: the domain-specific language DecL. DecL provides a user-friendly way to specify intricate, real-world distributions, employing a straightforward and human-readable syntax. It accommodates a wide range of scenarios, such as mixed severity curves, frequency parameter uncertainty, policy limit and attachment schedules, and multi-layer occurrence and aggregate reinsurance structures. Importantly, DecL's specification centers on quantities directly relevant to the user, like the mean loss or the frequency mixing coefficient of variation (CV), rather than obscure parameters. When parameters are utilized, they're separated into shape, scale, and location categories instead of being denoted by ambiguous Greek letters – avoiding struggles with alternative parameterizations of the gamma distribution, for example. Using DecL, `Aggregate` separates a distribution's specification from its numerical estimation, providing a succinct and user-friendly representation of complex real-world compounds. More details on DecL can be found in *The Grammar of Distributions*.

Estimating compound distributions is computationally intensive and challenging. Here, the Fast Fourier Transform (FFT) based algorithm offers significant advantages over competing methods, and it has been widely adopted in other fields. However, in actuarial circles, its application has not been commensurate with its potential utility, possibly due to its complexity. Albrecher *et al.* [2017], a technical text on reinsurance, says

> This [FFT] method is nowadays the fastest available tool to determine total claim size distributions numerically, but the implementation of the algorithm is not straightforward…

Implementation involves intricate calculations with complex numbers, which can intimidate users. Furthermore, key parameters must be selected carefully. The bandwidth is particularly challenging to select robustly. The FFT-based method can also fail unexpectedly, giving apparently nonsensical results. It is not straightforward to implement in

a spreadsheet, limiting accessibility for users reliant on such tools. Moreover, the FFT solution is hard to test in a traditional written exam, making it less suitable for inclusion in exam syllabi and hindering its broader adoption. These challenges led to its omission from the latest edition of Klugman *et al.* [2019], a prominent textbook in the field, despite thorough coverage in the first edition. `Aggregate` overcomes these obstacles and aims to establish the FFT method as the mainstream solution for estimating compound distributions by offering a user-friendly yet powerful and flexible Python [Van Rossum and Drake, 2009] implementation. To achieve this aim, it introduces two further innovations.

- `Aggregate` validates the FFT-based algorithm output by comparing the first three moments of its approximations against their theoretical values for each requested compound. If there is a close agreement, `Aggregate` reports that the approximation is "not unreasonable". Otherwise, it flags how it may fail to be reasonable. `Aggregate` computes the theoretical moments very accurately, either analytically or using numerical integration, and entirely independently of the FFT workflow. Validation testing gives the user confidence in the results. See *Validation* for the details.
- By offering reasonable default values for all parameters, `Aggregate` tackles a significant hurdle to the widespread adoption of FFT-based methods, what Parodi [2015] calls the need for "artful" parameterization. Among the parameters, the bandwidth used for discretization is particularly sensitive and challenging to select robustly. `Aggregate` puts substantial effort into its optimal selection, balancing the different forces that drive it to be small and large. See *Estimating the Bandwidth* for details. Users can accept the defaults or override them based on their specific needs.

In addition to these three principal innovations, the `Aggregate` implementation offers several other features.

- Severity distributions can be selected from any zero, one, or two shape parameter `scipy.stats` [Virtanen *et al.*, 2020] continuous distribution or entered as a discrete distribution (e.g., a sample of losses). Continuous distributions are described using the `scipy` shape-scale-location paradigm and are created by name, with no additional coding. `scipy.stats` currently supports over 100 continuous distributions.
- The ability to create, store, and reuse complex mixed severity distributions, including mixtures combining different parametric forms, is valuable for actuarial teams, particularly in reinsurance pricing. This functionality enables teams to share benchmark curves, enhancing efficiency and consistency in their analyses. By providing an easy way to generate and manage these distributions, `Aggregate` allows actuaries to streamline their workflows and facilitates knowledge exchange within their organizations.
- Access to a broad range of frequency distributions, including the fixed, Poisson, Bernoulli, binomial, geometric, logarithmic, negative binomial, Neyman A, and Pascal, along with mixed Poisson distributions with shifted gamma, shifted inverse Gaussian, beta, and Sichel's generalized inverse Gaussian mix distributions. Zero truncated and zero modified distributions are also available.
- The ability to directly enter empirical frequency and severity distributions, for example, to match an observed distribution or resample from an empirical distribution. This approach is handy for creating simple examples to use in teaching.
- The ability to apply policy limits and attachments and a full limits profile, optionally with shared frequency mixing, allows `Aggregate` to perform excess of loss exposure rating [Ludwig, 1991, Parodi, 2015].
- The ability to overlay complex per-occurrence and aggregate reinsurance towers with co-participations. `Aggregate` can model the effect of specific and aggregate covers, compute Table L and Table M charges, and solve related problems in reinsurance and large account pricing [Bahnemann, 2015, Fisher *et al.*, 2017].
- The ability to overlay aggregate variable features and compute the impact and value of sliding scale commissions, profit commissions, aggregate deductibles and limits, and price swing rated programs [Bear and Nemlick, 1990, Clark, 2014].
- A built-in library of over 150 examples illustrating these capabilities. Each can be created by name, or they can be built in a batch.
- Output is delivered using standard `pandas DataFrames` [McKinney, 2010, pandas development team, 2020], giving access to its powerful data manipulation and visualization tools.
- Built-in statistical functions such as the cumulative distribution `cdf`, survival `sf`, and probability density and mass `pdf`, `pmf`.

- Built-in estimation of quantiles or VaR, `q`, and TVaR, `tvar`. The computations explicitly allow for the discrete approximation used [Rockafellar and Uryasev, 2002]. These statistics are important in risk management and catastrophe risk evaluation.
- Built-in calculation of the probability of eventual ruin as a function of the premium rate and starting surplus for a compound Poisson process. This calculation uses the Pollaczeck-Khinchine (PK) formula [Asmussen and Albrecher, 2010, Embrechts *et al.*, 2013, Kyprianou, 2014, Mildenhall and Major, 2022]. The PK formula probabilities equal the distribution of a compound with a geometric frequency and the integrated original severity, so they can be computed using FFTs! The well-known Cramer-Lundberg formula is an approximation to the PK formula for thin-tail severity.
- Built-in visualization, leveraging `matplotlib` [Hunter, 2007] and `pandas` functions.
- The ability to compute technical (risk-loaded) premiums using spectral risk measures at different capital standards [Mildenhall and Major, 2022].

Combined with the flexibility, scalability, and performance of the underlying Python libraries, these features make `Aggregate` a versatile tool for actuarial professionals, researchers, and educators alike.

2.2 The `Aggregate` Calculation

The `Aggregate` tool calculates the probability distribution of total subject losses from a portfolio of insurance risks using the collective risk model [Klugman *et al.*, 2019]. This section defines the insurance-related terminology necessary to specify the distribution mathematically. Given its variation across markets and geographies, insurance terminology can be confusing. We aim to align our terminology with standard usage [Albrecher *et al.*, 2017, Mitchell-Wallace *et al.*, 2017, Parodi, 2015]. Once defined in this section, these terms are applied consistently throughout the paper without further elaboration.

Let X_i be a sequence of independent, identically distributed (iid) random variables modeling **ground-up** losses from individual occurrences. Ground-up losses are subject to insurance coverage but are before the application of any financial structures, such as contract limits and deductibles.

Insurance provides for coverage up to a per-occurrence limit y excess of a deductible a, paying

$$Z = \min(y, \max(X - a, 0))$$

against a ground-up loss X. Coverage is called ground-up if $a = 0$, excess if $a > 0$, and unlimited if $y = \infty$. The deductible is also known as the retention, priority, or attachment, hence a. The limit in an excess cover is often called the layer, hence y.

The distribution of Z usually has a mass at zero equal to $\Pr(X \le a)$, complicating analysis. To remove it, let $Y = (Z \mid X > a)$ equal the insured (layer) loss payment, conditional on a payment being made. Severity is conditional in this way if an attachment is specified. If no attachment is specified, $Y = X$ is used directly with no conditioning. Y is called **gross loss**. Gross loss feeds validation and, depending on reinsurance options, the compound frequency-severity convolution. The effect of these transformations on the ground-up cdf and sf are simple exercises in conditional probability that are spelled out in Klugman *et al.* [2019], Parodi [2015].

Let N be a counting distribution, independent of X_i, modeling the number of gross loss payments, i.e., the number of losses with $Y_i \ge 0$. `Aggregate` computes the compound distribution of gross losses

$$A = Y_1 + \cdots + Y_N.$$

Further, `Aggregate` can apply occurrence and aggregate reinsurance to A. Occurrence reinsurance provides coverage transforming gross loss Y into either ceded or retained (net) losses

$$\begin{aligned} C &= r \min(r_y, \max(Y - r_a, 0)) \\ R &= Y - C. \end{aligned}$$

Here r_y is the reinsurance occurrence limit, r_a the attachment, and r the share of the cover purchased, $0 \le r \le 1$. If $r_a > 0$ the reinsurance is excess, otherwise it is ground-up. If $r_a = 0$ and $r_y = \infty$ the cover is called a quota share.

If $r < 1$, the reinsurance is said to be partially placed or co-reinsured. `Aggregate` can model compound ceded or retained losses

$$A_C = C_1 + \cdots + C_N$$
$$A_R = R_1 + \cdots + R_N.$$

These distributions are not conditional on attaching the reinsurance, so A_C often has a mass at zero. If there is no occurrence reinsurance then $A_C = 0$ and $A_R = A$.

We call the **subject loss** for the compound whichever of ground-up (no limit and attachment), gross (no reinsurance), ceded, or net the user selects.

Lastly, aggregate reinsurance can be applied to A_C or A_R, transforming them in the same way as occurrence covers.

Both kinds of reinsurance can be stacked into multi-layer programs. See *The Reinsurance Clauses* for more details.

2.3 Installation and Reproducibility

`Aggregate` is part of the `aggregate` open-source Python package. `aggregate` also includes classes that support modeling with multiple compound distributions in a portfolio and has extensive pricing and capital allocation capabilities. For example, it was used to create all of the tables and figures in the book Mildenhall and Major [2022]. The source code is available at https://github.com/mynl/aggregate. The package is available from the PyPi package repository at https://pypi.org/project/aggregate/ and can be installed using:

```
pip install aggregate
```

There is extensive documentation [Mildenhall, 2023]. The latest version is hosted at https://aggregate.readthedocs.io/

We have included the Python code to reproduce each example throughout the paper. Once `aggregate` is installed, these code samples can be cut and pasted into Python.

2.4 Example

To fix ideas, here is an example using `Aggregate` to approximate and visualize a compound distribution and to compute various statistics. It is presented as an actuarial exam question. The answer shows the Python solution first, followed by a brief explanation. Amounts should be considered in thousands of USD, EUR, or GBP.

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

- Premium equals 750, and the expected loss ratio equals 67.5%.
- Ground-up severity has been fit to a lognormal distribution with a mean of 100 and CV of 500% ($\sigma = 1.805$).
- All policies have a limit of 1000 with no deductible or retention.
- Frequency is modeled using a Poisson distribution.

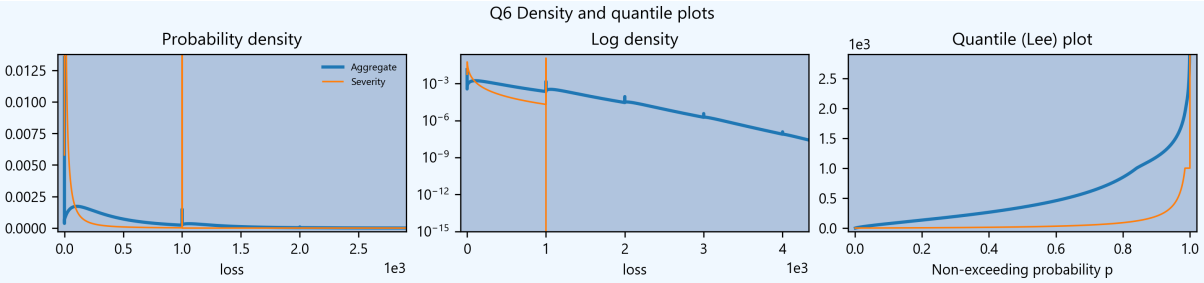
Question. Model aggregate losses using the collective risk model and compute:

1. The expected insured severity and expected claim count.
2. The expected value, standard deviation, CV, and skewness of aggregate losses.
3. The probability aggregate losses exceed the premium.
4. The expected value of aggregate losses limited to 2500, and the expected policyholder deficit ratio in excess of 2500.
5. The 99% and 99.9% VaR and TVaR of losses.
6. Plot the density (probability mass function) and quantile function for the distribution of total losses.

Answer.

All the functionality needed to solve this problem is accessed through the `build` function and the `qd` (quick display) helper function. The following Python code contains the answers.

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

In [2]: t = build('agg Trucking 750 premium at 0.675 lr 1000 xs 0 '
   ...:           'sev lognorm 100 cv 5 poisson')
   ...:

In [3]: qd(t)  # Qu 1 and 2

       E[X] Est E[X]    Err E[X]   CV(X) Est CV(X)  Skew(X) Est Skew(X)
X
Freq 6.3884                       0.39564            0.39564
Sev  79.245   79.245 -5.8027e-09  2.1191    2.1191    3.825       3.825
Agg  506.25   506.25 -5.8039e-09 0.92708   0.92708   1.5644      1.5644
log2 = 16, bandwidth = 1/8, validation: not unreasonable.

In [4]: print(f'\nQ3: Pr(Loss > premium) = {t.sf(750):.3f}')

Q3: Pr(Loss > premium) = 0.234

In [5]: qd(t.density_df.loc[[2500], ['F', 'S', 'lev', 'epd']]) # Qu 4

             F         S    lev       epd
loss
2500.0 0.99723 0.0027671 505.33 0.0018158

In [6]: qd(f'\nQ5: VaR 99% {t.q(0.99):,.0f}, 99.9% {t.q(0.999):,.0f}; '
   ...: f'TVaR 99% {t.tvar(0.99):,.0f}, 99.9% {t.tvar(0.999):,.0f}\n')
   ...:

Q5: VaR 99% 2,082, 99.9% 2,847; TVaR 99% 2,409, 99.9% 3,166


In [7]: t.plot()

In [8]: t.figure.suptitle('Q6 Density and quantile plots');
```



The compound distribution is specified using DecL in line 2. The program is a clean translation of the question data, and its meaning should be broadly self-evident. The `build` command insulates the user from the programming mechanics and tricky parameter selection by providing reasonable problem-specific defaults. It computes the expected conditional severity for the layer 1000 xs 0 and divides it into the loss pick (product of premium and loss ratio) to determine the expected claim count. It then uses an FFT-based algorithm to calculate a numerical approximation to the compound distribution. The approximation is stored in the pandas dataframe `t.density_df`.

The function `qd(t)` in line 3 displays `t.describe`, a dataframe that provides important validation information about the first three moments:

- Exact `E[X]` and estimated `Est E[X]` frequency, severity, and aggregate statistics in columns 1 and 2.
- Relative errors `Err E[X]` for the means, in column 3.

- Coefficient of variation `CV(X)` and estimated CV, `Est CV(X)` in columns 4 and 5.
- Skewness `Skew(X)` and estimated skewness, `Est Skew(X)` in the last two columns.

The line below the table shows `log2`, the log base-2 of the number of buckets used, and the bandwidth used in discretization. In this case, there are 2^{16} buckets of size $1/8 = 0.125$.

These statistics make it easy to test if the numerical estimation may be invalid: significant errors in the mean, second (CV), or third (skewness) moments are red flags. In this case, the moments all match to five significant digits, and the last item printed `validation: not unreasonable` shows the model did not fail any validation tests. Validation is interpreted like a null hypothesis – you can't "pass". The validation table provides the answers to questions 1 and 2:

- Insured severity equals 79.245 (less than the ground-up mean because of the 1000 limit).
- The expected claim count is 6.3884.
- The aggregate statistics are shown in the third row.

As input, expected losses equal 67.5% of 750 premium, 506.25.

The survival function `t.sf` answers question 3. The dataframe `t.density_df` contains columns for limited expected values, expected policyholder deficit indexed by loss level, and other values. Querying it answers question 4. The quantile and TVaR functions `t.q` and `t.tvar` answer question 5.

Lastly, the requested plot is produced using the `t.plot()` method. The density (left) and log density (center) show severity (orange) and compound losses (blue). The severity distribution has a mass at the policy limit, and the compound distribution has masses at zero (no claims) and at multiples of the limit. These are easier to see on the log density plot. The mixed nature of this type of compound has been noted before [Hipp, 2006] but the accuracy of FFT-based methods makes it especially apparent.

2.5 Context and Literature

The rest of the introduction puts `Aggregate` in context with other computational strategies in the literature and other available software packages.

Computational Strategies

There are five principal approaches to estimating compound distributions in the literature.

1. Simulation, Daykin *et al.* [1994], Klugman *et al.* [2019].
2. Panjer recursion, Panjer [1981], Bühlmann [1984], Daykin *et al.* [1994], Embrechts and Frei [2009], Klugman *et al.* [2019]. Panjer recursion applies to frequency distributions whose probabilities can be computed recursively as $p_k = (a + b/k)p_{k-1}$, a set that includes many commonly used frequencies.
3. FFT-based methods, Bertram [1981], Bühlmann [1984], Hürlimann [1986], Wang [1998], Grübel and Hermesmeier [1999], Mildenhall [2005], Embrechts and Frei [2009], Shevchenko [2010]. The details of the FFT method are presented in *Methodology and Algorithms*.
4. Fourier transform numerical integration methods, Bohman [1969], Bohman [1974], Heckman and Meyers [1983], Shevchenko [2010].
5. Approximations that use moment-matched distributions to the compound distribution or a transformation thereof, Daykin *et al.* [1994].

Simulation is the actuary's go-to method: it is easy, intuitive, and flexible but could be faster and more accurate. In many situations, its flexibility trumps other considerations. However, having more accurate and faster methods for certain applications is preferable, including individual large account pricing, reinsurance pricing, and catastrophe risk management.

Panjer recursion has the advantages of simplicity and accuracy: it is straightforward to program into a spreadsheet, is essentially exact, and easy to test in an exam. It is also possible to quantify its numerical properties. However, to compute n probabilities takes $O(n^2)$ operations, which limits its application to relatively small expected claim counts.

Insurance applications of FFT-based methods originated with Bertram [1981]. The algorithm takes $O(n \log(n))$ operations, giving it a substantial speed advantage for larger n. Like Panjer recursion, FFT-based methods are essentially exact, once the severity distribution has been discretized. FFT-based methods can be applied to any frequency distribution, whereas Panjer recursion is limited to those with recursive probabilities. Bühlmann [1984], Embrechts and Frei [2009] compare Panjer recursion and FFT-based methods, with the latter concluding it has a "tremendous timing advantage for a large number of lattice points".

Fourier transform numerical methods invert the characteristic function of the compound distribution at specific points using numerical integration, a delicate operation because the integrand oscillates rapidly. The FFT method is a generalization of this method since the inverse FFT is an approximation to the required integral. FFT-based methods evaluate the distribution at multiple points rather than one point at a time.

Approximation methods were popular before actuaries had easy access to computers. `Aggregate` provides normal, gamma, shifted gamma, lognormal, shifted lognormal, and maximum entropy distribution approximations. However, since very accurate approximations are now easy to compute, these methods are included only for historical and academic interest and are not considered further.

We contend that FFT-based methods perform at least as well as competing methods in all situations. Today's computers can discretize with tens of millions of buckets, although $2^{16} = 65536$ buckets generally produce accurate results. We have used FFTs to reproduce examples from numerous papers and examples accurately and have yet to find an example where it performs inadequately. It should be adopted as the standard method in actuarial work; never should the lognormal be used as an approximation again!

Implementation

Grübel and Hermesmeier [1999] provides a meticulous analysis of how the FFT-based algorithm works. We reprise the essentials in *Methodology and Algorithms*.

FFT-based methods are susceptible to aliasing, where the probability of large losses "wraps around" and appears near the origin, polluting the probabilities of small losses. Grübel and Hermesmeier [1999] and Shevchenko [2010] describe the use of exponential tilting to reduce aliasing error. Exponential tilting is the same method used in exponential family distributions to adjust the mean, and it is also used in large deviation theory. Alternatively, Wang [1998] describes how to pad input vectors as a more straightforward way to control aliasing. We find that padding almost always suffices to control aliasing, although `Aggregate` offers the option to apply exponential tilting if desired.

Embrechts *et al.* [1993], Grübel and Hermesmeier [2000] explain how to use Richardson extrapolation to improve density estimates. This method is easy to implement in Python.

Homer and Clark [2003] and Mildenhall [2005] describe using two-dimensional FFTs to model aggregates with bivariate frequency (for two different lines) and bivariate severity (net and ceded).

Non-Insurance Applications

The FFT-based 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 to control aliasing errors. They are interesting because they include modeling with high expected claim counts and thick-tailed severity with no mean.

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 with a closed-form 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 approach to accurate pairwise convolution that splits each component to limit the ratio of its most and least (non-zero) likely outcome. They also provide helpful estimates for assessing relative error and determining when an FFT estimate can be trusted.

Fourier Transform Literature

The concepts underlying Fourier transforms are used by different disciplines with different names and slightly different conventions, which can be confusing. Probability-oriented references talk about characteristic functions and do not scale by 2π, instead computing $t \mapsto \mathsf{E}[e^{itX}]$ see: Loeve [1955], Feller [1971], Lukacs [1970], Billingsley [1986], Malliavin *et al.* [1995], McKean [2014]. Analysis-oriented references on Fourier analysis and Fourier transforms use $t \mapsto \mathsf{E}[e^{-2\pi itX}]$, see: Stein and Weiss [1971], Stein and Shakarchi [2011], Strang [1986], Terras [2013], Körner [2022]. Malliavin is a nice treatment blending Fourier analysis and probability.

Similar Software

Kaas *et al.* [2008] and Parodi [2015] present the FFT-based algorithm in R. The `actuar` R package [Dutang *et al.*, 2008] supports calculating compound distributions using Panjer recursion, approximations, and simulation. The GEMAct Python package [Pittarello *et al.*, 2022] provides the same functionality in Python and includes FFT-based methods.

Competing implementations are more distribution and theory-focused rather than user-focused. For example, distributions are entered using oddly-named parameters rather than the mean and CV. They do not include any validation testing nor a means to estimate an appropriate bandwidth. They dedicate thousands of lines of code to implement discrete and continuous probability distributions from scratch. In contrast, `Aggregate` leverages `scipy.stats` for this basic functionality and only adds what is missing: probability generating and a moment functions for each frequency distribution, and a sub-class of `scipy.stats.rv_continuous` to handle limits, attachments, and conditioning. Almost all the code in `Aggregate` is domain-specific.

An excellent example of the philosophical difference between `Aggregate` and similar tools is discretization. GEMAct, `actuar`, and `Aggregate` include a function to discretize a severity distribution in various ways. But in `Aggregate`, this function is behind the scenes, never called directly by the user. Discretization is invoked automatically to accomplish a higher-level task the user requires.

There are also several commercial tools available that simulate compounds. These include ReMetrica, MetaRisk, Tyche Capital Model, and @Risk. These are general-purpose tools to build economic capital models, not just compound distributions. `Aggregate` can fit into a broader capital modeling workflow by providing accurate and realistic compound distribution samples or densities for individual modeling units.

Intended Purpose

The functionality of `Aggregate` matches its intended purpose in the actuarial analysis workstream: to generate accurate compound distributions *once all distribution parameters are known*. As a result, it does not include tools to estimate loss picks or fit or recommend frequency distributions or severity curves. It can maintain benchmark severity curves for shared use.

2.6 Overview

The remainder of the paper is structured as follows. The next section, *The Grammar of Distributions*, describes DecL. Section 4 details the *Methodology and Algorithms* and Section 5 covers *Examples and Workflow*.

3 The Grammar of Distributions

The first challenge in computing a compound distribution lies in specifying it succinctly yet flexibly, essentially defining the grammar of distributions. Traditional approaches expose Application Programming Interfaces (APIs) for frequency, severity, reinsurance, and aggregate objects, requiring the user to assemble the parts manually. However, this approach prioritizes implementation details and mathematical parameterizations over meaningful concepts and quantities that hold practical relevance to the user. For instance, GEMAct [Pittarello *et al.*, 2022] follows this approach. While API-based solutions are suitable for constructing simple textbook distributions, they often fall short when confronted with a real-world problem, limiting their usability and scalability.

In contrast, `Aggregate` takes a different approach by prioritizing risk and insurance concepts. It introduces DecL, a domain-specific language designed explicitly for specifying compound distributions. DecL allows users to express complex distributions using a straightforward and intuitive syntax, enabling a more direct and meaningful representation of real-world scenarios. At the same time, DecL specifies simple compounds with minimal overhead, which eases the learning curve for a new user. By focusing on concepts and quantities relevant to risk and insurance analysis, `Aggregate` provides a more efficient and intuitive process for specifying a compound distribution.

To demonstrate the power of DecL, consider specifying a compound distribution to model losses from policies written at different limits and attachments, ceded to an excess of loss reinsurance program. This model incorporates a mixed severity and a mixed Poisson frequency with a shared mixing distribution. In real-world scenarios, constructing such a compound distribution can involve many components and entries. For instance, industry-standard mixed exponential distributions used in the US often comprise 5 to 10 parts. At the same time, a limits profile may contain dozens (in the case of liability) to hundreds (in the case of property) of entries. The resulting outer product can sum up hundreds or even thousands of individual mixed Poisson compounds. With the help of DecL, specifying such a model with n limit and attachment bands, m severity components, and r occurrence reinsurance layers is straightforward, as the following code snippet demonstrates. The code's meaning is revealed through the remainder of this section.

```
agg ProfileExample
    [premium_1 ... premium_n] premium at [lr_1 ... lr_n] lr
    [limit_1 ... limit_n] xs [attach_1 ... attach_n]
sev
    [scale_1 ... scale_m] * expon wts [wt_1 ... wt_m]
occurrence ceded to
    [layer_1 and ... and layer_r]
mixed gamma 0.25
```

The name DecL (Dec Language) derives from an insurance policy's declarations ("dec") page that spells out key coverage terms and conditions. A reinsurance slip performs the same functions. DecL is designed to make it easy to go from "dec page to distribution." The rest of this section describes domain-specific languages and then details the DecL grammar for specifying a compound distribution.

3.1 Domain-Specific Languages

The use of domain-specific languages in software development is recognized for its trade-off between generality and expressiveness within a specific domain [Mernik *et al.*, 2005]. They can improve productivity and broaden adoption by leveraging well-known domain-specific notations and improve verification, analysis, and error reporting, among other advantages. In the case of `Aggregate`, the domain-specific language DecL is employed to leverage a natural, insurance-specific terminology familiar to users.

`Aggregate` parses and translates user-specified DecL code into traditional API calls, which are then executed by the relevant underlying classes in the software. This approach shields the user from the intricacies of the underlying object-oriented class structure and eliminates the need to remember individual argument names. DecL supports multiple input paradigms and automatically detects inconsistent inputs, providing a more user-friendly experience.

The benefit of using DecL is evident when comparing concise and readable DecL code examples to more verbose and complex underlying API calls. For instance, the *Example*

```
agg Trucking 750 premium at 0.675 lr 1000 xs 0 sev lognorm 100 cv 5 poisson
```

demonstrates how a one-line DecL code snippet succinctly defines a compound distribution, whereas the equivalent API call requires more forbidding code.

```
from aggregate import Aggregate

a01 = Aggregate(name='Trucking', exp_premium=750., exp_lr=0.675,
                exp_attachment=0.0, exp_limit=1000.0, sev_name='lognorm',
                sev_mean=100.0, sev_cv=5., freq_name='poisson')
```

While DecL simplifies the workflow for users, programmers still have the flexibility to access the underlying class API directly when needed. This ensures that advanced users can leverage the full capabilities of `Aggregate`.

3.2 Specifying a Compound Distribution in DecL

The syntax for a complete DecL compound specification has eight clauses

```
agg <NAME> <EXPOSURE> <LIMIT*> <SEVERITY> <OCC_RE*> <FREQUENCY> <AGG_RE*> <NOTE*>
```

that relate back to *The Aggregate Calculation*. Throughout this section, terms in `<UPPER_CASE>` represent user inputs, while lowercase terms refer to language keywords. The name, exposure, severity, and frequency clauses are required, while the limit, occurrence and aggregate reinsurance, and note clauses are optional (denoted by an asterisk). The following subsections examine each clause in detail.

Before delving into specific details, a few general notes on DecL are worth mentioning. DecL is case-sensitive and operates on single-line programs. While we sometimes use line breaks for readability, they must be removed before executing the `build` function or combined with a Python `\` newline continuation.

DecL accepts numbers in percent notation (`67.5% lr`), as well as in regular or scientific notation. A minus sign joined to a number, like -1, has a different effect than $-\ 1$ with a space, as explained in *The Severity Clause*. DecL supports three arithmetic operations: division, exponentiation, and the exponential function. These are sufficient to express probabilities as fractions and to calculate the `scipy.stats` scale for a lognormal distribution as `exp(mu)/exp(sigma**2/2)`. Python's `f`-string format lets you inject variables directly into DecL programs, such as `f'sev lognorm {x} cv {cv}'`, eliminating the need for extensive mathematical functionality.

The Name Clause

The name clause, **agg** `<NAME>` declares the start of a compound distribution using the **agg** keyword. Keywords are part of the language, like **select** in SQL. Other parts of the `aggregate` package define **port** and **distortion** keywords to create portfolio and spectral distortion objects. `NAME` is a label, such as `Trucking`. It can contain letters, numbers, periods, underscore, and dashes, but it must start with a letter.

The Exposure Clause

The exposure clause determines the volume of insurance. Volume can be stipulated explicitly as the expected loss or implicitly as the expected claim count or a claim count distribution. The clause has five forms: the first three are explicit, the fourth inputs the expected claim count, and the last the count distribution.

```
<EXP_LOSS> loss
<PREMIUM> premium at <LR> lr
<EXPOSURE> exposure at <RATE> rate

<CLAIMS> claims

dfreq <OUTCOMES> <PROBABILITIES>
```

- **loss** is a keyword and `EXP_LOSS` equals the loss pick. The claim count is derived by dividing by average severity. It is typical for an actuary to estimate the loss pick and select a ground-up severity curve, and then derive frequency in this way.

- Similarly **premium**, **at**, **lr**, **exposure**, and **rate** are keywords to enter expected loss as premium times a loss ratio, or exposure times a unit rate. Actuaries often take plan premiums and apply loss ratio picks to determine expected losses rather than starting with a loss pick. Underwriters sometimes think of benchmark unit rates (e.g., per vehicle, per 100 insured value, per location).
- **claims** is a keyword. The `s` is optional, allowing `10 claims` and `1 claim`. Expected loss equals expected claim count times average severity.
- The discrete distribution syntax directly specifies frequency outcome and probability vectors, as described next.

Discrete Distributions

Discrete frequencies and severities can be specified using the keywords `dfreq` and **dsev** (see *The Severity Clause*). There are some special rules for discrete distributions that we gather together here. The general form is

```
dfreq <OUTCOMES> <PROBABILITIES>
dsev  <OUTCOMES> <PROBABILITIES>
```

For example, specifying outcomes `[1 2 3]` and probabilities `[0.75 3/16 1/16]` means there is a 3/4 chance of an outcome of 1, a 3/16 chance of 2, and so forth. If all outcomes are equally likely, the probability vector may be omitted. Commas are optional in vectors, and only division arithmetic is supported. Outcome ranges can be defined using the Python slice syntax `[1:6]` for `[1 2 3 4 5 6]` or `[0:50:25]` for `[0 25 50]`. Note that, unlike Python, the last element is included. Non-positive outcomes are replaced by zero, but there should be none, given the context. The outcomes need not be distinct or sorted. `Aggregate` accumulates the probability by outcome before using the distribution. Thus, a sample of losses or observed claim counts can be input and used directly. This syntax is handy for constructing simple examples and using empirical distributions. Entering `dfreq [1]` denotes one claim with certainty, which reduces the compound to the severity, and **dsev** `[1]` is a loss of one with certainty, reducing to the frequency.

The Limit Clause

The optional limit clause specifies a per-occurrence limit and deductible, with

```
<LIMIT> xs <DEDUCTIBLE>
```

replacing ground-up severity X with gross loss $Y = \min(y, \max(X - a, 0)) \mid X > a$.

The Severity Clause

The severity clause specifies the ground-up size of loss distribution. Losses are conditional on attaching the layer when the limit clause is present. The severity clause can be specified in three ways. The first form enters the mean and CV of the unlimited ground-up severity. This form can be used for distributions with only one shape parameter and the beta distribution on $[0, 1]$. `aggregate` uses a formula (lognormal, gamma, beta) or a 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 enters the shape parameters directly, and the third defines a discrete distribution analogously to `dfreq`.

```
sev <DIST_NAME> <MEAN> cv <CV>
sev <DIST_NAME> <SHAPE1> <SHAPE2>
dsev <OUTCOMES> <PROBABILITIES>
```

- **sev** is a keyword indicating the severity specification.
- `DIST_NAME` is the `scipy.stats` continuous random variable distribution name.
- `MEAN` is the expected loss.
- **cv** (lowercase) is a keyword indicating the entry of the CV, to distinguish from inputting shape parameters.
- `CV` is the loss coefficient of variation.

- <SHAPE1>, <SHAPE2> are the required scipy.stats shape variables for the distribution.
- **dsev** is a keyword to create discrete severity. It works in the same way as dfreq, see *Discrete Distributions*.

DIST_NAME can be any scipy.stats continuous distribution with no shape parameters (e.g., norm, Gaussian normal; unif, uniform; and expon, the exponential), one shape parameter (e.g., **pareto**, **lognorm**, **gamma**, **invgamma**, **loggamma**, and **weibull_min** the Weibull), or two (e.g, **beta** and **gengamma**, the generalized gamma). Shape parameters entered for zero parameter distributions are ignored. See [Mildenhall, 2023] Section 2.4.4.5 for a full list of available distributions.

The **dsev** form can be used to re-sample from a set of observed losses: simply pass the losses in as OUTCOMES. They can be in any order and do not need to be distinct. The software automatically summarizes them by outcome and accumulates the probabilities.

A parametric severity clause can be transformed by applying scaling and location factors, following the scipy.stats loc (for a shift or translation) and scale syntax. The syntax and examples are as follows.

```
sev <SCALE> * <DISTNAME> <SHAPE1> <SHAPE2> + <LOC>
sev <SCALE> * <DISTNAME> <SHAPE1> <SHAPE2> - <LOC>
```

- **sev lognorm 10 cv 3** creates a lognormal with mean 10 and CV 3.
- **sev 10 * lognorm 1.517** creates a lognormal, $10X$, $X \sim$ lognormal$(\mu = 0, \sigma = 1.517)$ with CV equal to $\sqrt{\exp(\sigma^2) - 1} = 3.0$.
- **sev 10 * lognorm 1.517 + 20** creates $10X + 20$.
- **sev 5 * expon** creates an exponential with mean (scale) 5; there is no shape parameter.
- **sev 5 * uniform + 1** creates a uniform with scale 5 and location 1, taking values between 1 and 6. The uniform has no shape parameters.
- **sev 50 * beta 2 3** creates $50Z$, $Z \sim \beta(2, 3)$ a beta with two shape parameters 2, 3.
- **sev 100 * pareto 1.3** creates a single parameter Pareto for $x \geq 100$ with shape 1.3 and scale 100.
- **sev 100 * pareto 1.3 - 100** creates a Pareto with survival function $S(x) = (100/(100 + x))^{1.3}$, :math:xge 0`.
- A negative location shift must be entered with a space **sev lognorm 1 - 10** rather than **sev lognorm 1 -10**. The latter appears to the parser as two shape parameters.

Ground-up losses can be conditioned to lie in a range $(lb, ub]$ by using the **splice** keyword. For example

```
sev lognorm 100 cv 0.25 splice [80 130]
```

creates a lognormal ground-up severity with mean 100 and CV 0.25 conditioned to $80 < X \leq 130$. Conditioning differs from a layer 50 **xs** 80 in two ways: losses lie in the range 80 to 130 rather than 0 to 50, and the result is a continuous distribution, whereas the layer has a probability mass at 50. Splicing helps create flexible mixtures, see *Vectorization*. It is implemented using Python decorators to adjust the pdf, cdf, sf, and quantile functions. This approach has no impact on performance when there is no conditioning.

Severity distributions created with the **sev** keyword are continuous unless the underlying scipy.stats distribution can take negative values, in which case they have a mass at zero corresponding to the probability of a non-positive loss. Severities created using **dsev** are discrete.

Understanding the interaction of mixed and discrete severities with the limit clause is crucial. If the limit clause is missing, gross loss equals ground-up loss with no adjustment. The result has a mass at zero when ground-up loss does. When there is a limit clause, the gross loss is conditional on a ground-up loss to the layer. As a result, a missing limit clause is subtly different from a layer inf **xs** 0. Here are two examples. The first uses the (continuous) normal as severity. All negative values are accumulated into the zero bucket, creating a mixed distribution. Consider:

```
agg NoLimitClause 1 claim          norm fixed
agg LimitClause   1 claim inf xs 0 norm fixed
```

When there is no limit clause, the gross loss is a 50/50 mixture of zero and a half-normal with mean $\sqrt{2/\pi} = 0.79788$, giving a gross mean of 0.39894. With a limit clause, the mass at zero is conditioned away, and the gross loss is a half-normal.

The second example uses a discrete distribution.

```
agg NoLimitClause 1 claim          dsev [-1 0 1 1 2] fixed
agg LimitClause   1 claim 10 xs 0 dsev [-1 0 1 1 2] fixed
```

`Aggregate` summarizes the input ground-up discrete distribution to outcomes 0, 1, and 2 with probabilities 2/5, 2/5, and 1/5. When there is no limit clause, gross loss equals ground-up, giving mean severity of 4/5. With a limit clause, gross loss equals ground-up conditional on a loss, meaning outcomes 1 and 2 with probabilities 2/3 and 1/3, giving mean severity 4/3. The behavior illustrated in these two examples ensures that `Aggregate` works as expected. Without it, a mass at zero defined by a discrete distribution would mysteriously disappear in simple examples.

Appending **`!`** to the severity clause makes gross losses unconditional on attaching the layer, altering the default behavior. To see the effect, consider the two DecL programs:

```
agg Conditional  1 claim 2 xs 1 dsev [1:3]   fixed
agg Uncond       1 claim 2 xs 1 dsev [1:3] ! fixed
```

Both programs have one claim for sure. The `Conditional` program uses severity $(X-1) \mid X > 1$, giving outcomes 1 and 2, and expected severity of 1.5. The `Uncond` program models 1 claim ground-up and then applies the limit and deductible, giving outcomes 0, 1, and 2, and expected severity of 1.

The Frequency Clause

The frequency clause completes the specification of the frequency distribution. If `dfreq` is used in the exposure clause, then there is nothing else to specify, and the frequency clause is unnecessary. Otherwise, the clause must contain the frequency distribution name and appropriate parameters.

There are two types of frequency distributions: basic named distributions falling into the Panjer (a, b, n) class [Klugman *et al.*, 2019], and mixed Poisson distributions. The basic frequency distributions are the fixed, Poisson, Bernoulli, binomial, geometric, logarithmic, negative binomial, Neyman A (Poisson-Poisson compound), and Pascal (Poisson-negative binomial compound). All take values $0, 1, \ldots$, except the logarithmic, which takes values $1, 2, \ldots$. These distributions are specified by name. For example,

```
agg NB 10 claims dsev [1] negbin 3
```

creates a negative binomial with mean 10 and variance multiplier (the ratio of variance to mean) of 3. This is an exception to the general rule that frequency `SHAPE1` inputs a CV.

Zero truncated and zero modified [Klugman *et al.*, 2019] versions of the Poisson, Bernoulli, binomial, geometric, negative binomial, and logarithmic are specified by appending **`zt`** or **`zm`** `p0`. For example,

```
agg NB 10 claims dsev [1] poisson zm 0.4
```

creates a zero modified Poisson with mean 10 and probability 0.4 of taking the value 0. The modified and truncated versions are implemented using Python decorators in an entirely generic manner, with essentially no additional coding. The basic frequency distributions are common in textbook, catastrophe modeling, and small portfolio applications.

The `fixed` frequency supports a fixed claim count when losses are specified directly: **`agg`** `Example 100` **`loss sev`** `dfreq [1:9] fixed`. In this case, the user must ensure that the expected frequency is an integer. Fixed frequency can also be input in the exposure clause as `dfreq [n]`.

Mixed Poisson frequency distributions, where $N \sim \text{Po}(nG)$ for a mixing distribution G with mean 1, are appropriate for modeling larger portfolios [Mildenhall, 2017]. These are specified

```
agg Mixed 10 claims dsev [1] mixed <DISTRIBUTION> <SHAPE1> <SHAPE2>
```

`SHAPE1` always specifies the CV of the mixing distribution, and given that, N has variance $n(1 + (cv)^2 n)$. The meaning of the second shape parameter varies by distribution. The following mixing distributions are supported.

- `mixed gamma <SHAPE1>` is a gamma-Poisson mix, i.e., a negative binomial. Since the mix mean (shape times scale) equals one, the gamma mix has shape $(cv)^{-2}$.
- `mixed delaporte <SHAPE1> <SHAPE2>` uses a shifted gamma mix. The second parameter equals the proportion of certain claims, which determines a minimum claim count. This distribution is useful to ensure the compound distribution does not over-weight the possibility of very low loss outcomes. A higher proportion of certain claims increases the skewness of the frequency and compound distributions.
- `mixed ig <SHAPE1>` has an inverse Gaussian mix distribution.
- `mixed sig <SHAPE1> <SHAPE2>` has a shifted inverse Gaussian mix, with parameter 2 as for the Delaporte.
- `mixed beta <SHAPE1>` is a beta-Poisson, where the beta has mean 1 and cv `<SHAPE1>`. Use with caution.
- `mixed sichel <SHAPE1> <SHAPE2>` is Sichel's (generalized inverse Gaussian) distribution with `<SHAPE2>` equal to the λ parameter [Johnson *et al.*, 2005].

It is worth noting that the negative binomial distribution can be entered in two ways. The first uses the `negbin` named frequency distribution and specifies the variance multiplier v. This approach is commonly used for small claim counts. The second involves using `mixed gamma` and setting the mixing CV. This approach is more suitable for larger claim counts. To reconcile these two methods, equate the frequency variance: $nv = n(1 + (cv)^2 n)$, where n equals the expected claim count. See *Vectorization* for more about mixed frequencies.

The Reinsurance Clauses

`Aggregate` can model multi-layer per-occurrence and aggregate reinsurance structures. Reinsurance is specified flexibly in a way that includes proportional (quota share) and excess of loss covers as special cases, unlike some other approaches that unnecessarily separate the two. For a comprehensive survey of reinsurance and its impact on transforming subject losses, see Albrecher *et al.* [2017], Parodi [2015].

An individual layer, for both occurrence and aggregate covers, is specified as

```
<<SHARE> so> <LIMIT> xs <ATTACHMENT>
```

`<SHARE> so` specifies a share (partial placement); **so** stands for "share of". The default share is 100% if this sub-clause is omitted. The layer transforms gross loss X into

$$s \times \min(y, \max(X - a))$$

where s, y, a are the share, limit, and attachment. Unlimited cover is entered using an infinite limit `inf`. With this notation, a 65% quota share is simply `65% so inf xs 0`. Severity is not conditional on attaching occurrence reinsurance covers (whereas it is for the limit clause), which is the expected behavior.

Multiple layers can be stacked together using the **and** keyword. For instance:

```
50% so 250 xs 250 and 90% so 500 xs 500 and 95% so inf xs 1000
```

models a 50% placement of a 250 xs 250 layer, 90% placement of 500 xs 500, and a 95% placement of unlimited excess 1000. The layers must not overlap, but they do not need to be contiguous. Each layer is applied separately to subject losses; they do not inure to each other's benefit. It is possible to model any non-decreasing function of underlying losses using multiple layers, capturing all reasonable indemnity functions [Huberman *et al.*, 1983].

The concepts described above apply to both occurrence and aggregate covers. Occurrence reinsurance is applied to individual claims and adjusts severity. It is usually specified before the frequency clause but can also be specified after a named compound. The compound can capture the cession or losses net of the cession. For example:

```
occurrence ceded to 900 xs 100
occurrence net of 75% so 750 xs 250
```

The first line models losses ceded to the 900 xs 100 layer creating a compound

$$A_C = \sum_{i=1}^{N} \min(900, \max(X_i - 100, 0)).$$

The second models losses net of a 75% placement 750 xs 250, creating a compound

$$A_R = \sum_{i=1}^{N} X_i - 0.75 \min(750, \max(X_i - 250)).$$

Aggregate reinsurance, on the other hand, is always specified after the frequency clause and is applied to total losses. It follows the same pattern as occurrence but uses the keyword **`aggregate`**. For example:

```
aggregate ceded to 50% so 4000 xs 1000
```

models the cession $0.5 \min(4000, \max(A - 1000, 0))$ where $A = \sum_{i=1}^{N} X_i$ represents the subject compound distribution.

Occurrence and aggregate programs can be combined to represent different views. The occurrence program insures to the benefit of the aggregate. For instance:

```
occurrence ceded to 500 xs 500 poisson aggregate net of 1000 xs 0
occurrence net of 90% so 750 xs 250 poisson aggregate net of 30% inf xs 0
```

The first row models cessions to 500 xs 500 with a 1000 annual aggregate deductible, while the second models the net result after an excess of loss and a 30% quota share on the net of excess losses.

All occurrence reinsurance is modeled with unlimited reinstatements. Cessions to a program with limited reinstatements can be modeled by combining occurrence and aggregate programs. For example, a 500 xs 500 layer with 3 reinstatements (four limits in total) is expressed as:

```
occurrence ceded to 500 xs 500 poisson aggregate ceded to 2000 xs 0
```

`Aggregate` cannot directly model net losses from a limited reinstatement excess of loss because doing so involves tracking both net without reinstatements and the impact of the reinstatement clause.

Excess underwriters often layer a large program into multiple smaller layers. Such a tower of layers can be specified by giving the layer breakpoints. For example,

```
occurrence net of tower [0 250 500 1000]
aggregate ceded to tower [0:10000:1000]
```

model 250 xs 0, 250 xs 250, and 500 xs 500 occurrence layers, and an aggregate program in bands of 1000 up to 10000, respectively.

A policy limit and deductible can be specified via a limit clause or a single occurrence reinsurance clause. As a general rule, the limit clause should be preferred for its simplicity and precision whenever feasible. While both yield identical results for ground-up covers, they exhibit a crucial difference for excess covers. In the case of a limit clause, losses are conditional upon attaching, whereas they are not for occurrence reinsurance. Consequently, the expected severity would be 1.5 in the first scenario below (outcomes 1 or 2) and 1 in the second scenario (outcomes 0, 1, or 2).

```
agg Limit-Clause 1 claim 2 xs 1 dsev [1:3]                          fixed
agg Reinsurance  1 claim        dsev [1:3] occurrence ceded to 2 xs 1 fixed
```

The limit clause and occurrence reinsurance clause also differ significantly in their implementation. The limit clause is implemented analytically and adjusts the cumulative distribution function (cdf) and survival function (sf) of the underlying `scipy.stats` continuous random variable. Its strength lies in precision, enabling its inclusion in validation, but it offers limited flexibility. For instance, it only accommodates a single layer and does not allow partial placements. On the other hand, the occurrence reinsurance clause is implemented numerically, allowing for greater flexibility, including partial placements and multiple layers. It modifies the discretized severity distribution after applying the limit clause. However, this flexibility makes it too complicated to include in the validation process easily.

We end the discussion of reinsurance by pointing out that occurrence covers require far more effort to model than aggregate ones. An occurrence cover adjusts severity and must then be passed through the FFT-based convolution algorithm. In contrast, aggregate covers are a straightforward transformation of the compound distribution.

The Note Clause

The optional note clause is `note{text of note}`. It serves two purposes. It provides a space to include a description or additional details about the compound distribution beyond its name. It can be useful for providing context or clarifying the purpose of the compound distribution. Secondly, it can be used to encode calculation parameters. By including specific parameters, users can customize the behavior of the calculation process. For example,

```
note{Prems/Ops 3 Severity, 2023 update; log2=17; normalize=False}
```

adds a curve description and specifies that the object should be calculated using the parameters `log2=17` and `normalize=False`. Semicolons separate different parts of the note and parameters are passed using the `key=value` syntax. For more detailed information on the available parameters and their usage within the note clause, see *Python and Aggregate Implementation*.

3.3 Vectorization

The power of DecL is enhanced through vectorization, which enables the simultaneous processing of multiple values for expected loss, claim counts, premium, loss ratio, exposures, unit rates, limits, attachments, and severity parameters. As mentioned in the introduction to this section, vectorization is a key feature of DecL that brings significant advantages. Before describing vectorization, we must present its mathematical basis.

Suppose there are r families of iid severity random variables $X^{(i)}$. Each represents a different type of business, or a different severity mixture component, or both. Family i has expected claim count n_i. Frequency for each family is a mixed Poisson $N^{(i)} \sim \text{Po}(n_i G)$ where G is a shared mixing distribution with mean 1. G is used to capture common effects across families, such as the impact of weather or economic activity on frequency. Compound losses for family i are

$$A_i = X_1^{(i)} + \cdots + X_{N^{(i)}}^{(i)}$$

If G is non-trivial, then shared mixing induces correlation between the family compounds A_i; otherwise they are independent. Standard results in the theory of compound Poisson processes [Klugman *et al.*, 2019] show that the total loss across all families

$$A = A_1 + \cdots + A_r$$

is a mixed compound Poisson compound with frequency $N \sim \text{Po}(nG)$, $n = \sum_i n_i$, and mixed severity with distribution

$$F_X(x) = \sum_i \frac{n_i}{n} F_{X_i}(x).$$

With that background, we can describe the two main applications of vectorization. First, it is used to specify a mixed severity distribution. For example, the widely used mixed exponential distribution [Corro and Tseng, 2021, Parodi, 2015, Zhu, 2011] is specified as:

```
sev [<MEAN_1> ... <MEAN_n>] * expon wts [<WT_1> ... <WT_n>]
```

The weights are applied to expected claim counts. When exposure is entered as expected loss, `Aggregate` automatically performs the slightly intricate calculations needed to determine the claim counts by component.

The exponential has no shape parameter. More generally, the distribution type and shape parameters can both be vectorized, an approach used in what Albrecher *et al.* [2017], Section 3.5, call splicing, where different distributions are used for small and large claims. For example,

```
sev [100 150] * [expon pareto] [1 2.5] + [0 -150] wts [0.8 0.2] splice [0 200 inf]
```

weights an exponential with mean 100 for losses between 0 and 200 with a Pareto with shape $\alpha = 2.5$ and scale 150 for loss above 200. The splice vector gives the range bounds for each distribution. Alternatively, splices can be specified with two vectors, giving the lower and upper bounds: `splice [0 200] [200 inf]`. The two conditional distributions have weights 0.8 and 0.2, respectively. There are two subtle points to note: a shape parameter for the exponential of 1 is added for clarity, but it is ignored, and the Pareto is shifted back to the origin by the location term rather than being a single parameter Pareto. Alternatively, a single parameter Pareto on losses above 200 is entered

```
sev [100 200] * [expon pareto] [1 2.5] wts [0.8 0.2] splice [0 200 inf]
```

The second vectorization application allows one compound distribution to model multiple units with a shared frequency mixing distribution. Here, a single DecL program can effectively capture the characteristics of multiple units by specifying the corresponding values in vector notation. The DecL code snippet:

```
agg MultiUnitExample
  [1000 2000 3000] premium at [.8 .75 .7] lr
  [1000 2000 5000] xs 0
sev
  lognorm [50 100 150] cv [0.1 0.15 0.2]
mixed gamma 0.4
```

models three units with premiums of 1000, 2000, and 3000 and expected loss ratios of 80%, 75%, and 70% from policies with limits of 1000, 2000, and 5000, where the units have lognormal severities with means 50, 100, and 150 and CVs 10%, 15% and 20%. The mixed Poisson frequency distribution shares a gamma mixing variable with a CV of 40% across all three units to induce correlation between them. The absence of a `wts` term distinguishes this from a mixed severity.

When vectorized exposures are combined with a mixed severity distribution, `Aggregate` automatically generates the relevant outer cross product of exposure and severity components. This form enables streamlined calculations for excess of loss reinsurance exposure rating, providing a concise and human-readable representation as demonstrated by the example in the introduction to this section.

Vectorized exposures and mixed severity distributions can only be used with mixed Poisson frequency distributions because they rely on the identity that a compound with a mixed severity has the same distribution as the sum of the compounds over each severity component.

3.4 Defining the DecL Grammar

The DecL grammar is specified in Backus Naur form and is interpreted using the SLY package [Beazely, 2022], a Python implementation of lex and yacc (yet another compiler-compiler), tools commonly used to write parsers and compilers. Parsing is based on the Look Ahead Left-to-Right Rightmost (LALR) algorithm. These concepts are explained in Aho *et al.* [1986], Levine *et al.* [1992].

The lexer defines tokens of the language, either as explicit strings, like the **agg**, **sev**, or `poisson`, or as patterns defined by regular expressions. For example, a string identifier is determined by the regular expression `[a-zA-Z][\.:~_a-zA-Z0-9\-]*`. See [Mildenhall, 2023] Section 4.2 for a full list of tokens.

The lexer breaks the input DecL program into tokens which are then passed to the parser, which converts them into a keyword-value dictionary understood by the object API. Grammar rules show how higher-level constructs are created from lower-level ones. For example, *The Exposure Clause* is written:

```
exposures  ::= numbers CLAIMS
             |  numbers LOSS
             |  numbers PREMIUM AT numbers LR
             |  numbers EXPOSURE AT numbers RATE
```

The grammar builds up various components until they are combined to specify the most general compound distribution. The complete grammar is laid out in [Mildenhall, 2023] Section 4.3.

4 Methodology and Algorithms

The previous section described how to specify a compound succinctly yet flexibly. We now move to the second challenge: estimating its distribution probabilities quickly and accurately. Specifically, we want a way to compute the distribution function of a compound random variable

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

where the (suitably limited and conditioned) severity random variables are independent and identically distributed, and all X_i are independent of the frequency (claim count) random variable N. We want to approximate F_A at equally spaced outcomes kb, $k = 1, 2, \ldots$. Using the tower rule for conditional expectations and the independence assumptions, we can derive the well-known formula

$$\begin{aligned} F_A(kb) :&= \Pr(A < kb) \\ &= \sum_n \Pr(A < kb \mid N = n) \Pr(N = n) \\ &= \sum_n F_X^{*n}(a) \Pr(N = n), \end{aligned}$$

where $F_X^{*n}(a)$ denotes the distribution of the sum of n independent copies of X. Usually, this problem has no analytic solution because the distribution of sums of X_i is rarely known. For example, there is no closed-form expression for the sum of two lognormals [Milevsky and Posner, 1998]. However, things are more promising in the Fourier domain because the characteristic function of F_X^{*n} is always known – it is simply the nth power of the characteristic function of F_X.

The characteristic function of A can be written in terms of the characteristic function of severity and the frequency probability generating function (pgf) $\mathscr{P}_N(z) := \mathsf{E}[z^N]$ using the same logic as for distributions:

$$\phi_A(t) := \mathsf{E}[e^{itA}] = \mathsf{E}[\mathsf{E}[e^{itA} \mid N]] = \mathsf{E}[\mathsf{E}[e^{itX}]^N] = \mathscr{P}_N(\phi_X(t)).$$

The pgfs of most common frequency distributions are known. For example, if N is Poisson with mean λ then $\mathscr{P}_N(z) = \exp(\lambda(z - 1))$.

Two things combine to make this identity useful. The first is Poisson's summation formula, which says (roughly) that the characteristic function of an equally spaced sample of a distribution equals an equally spaced sample of its characteristic function. The second is the existence of the Fast Fourier Transform algorithm, which makes it very efficient to compute and invert characteristic functions of equally spaced samples. Using Poisson's formula, and using FFTs in steps 2 and 4, supports the following FFT-based algorithm to estimate compound probabilities:

1. Discretize the severity cdf to approximate F_X.
2. Apply the FFT to approximate ϕ_X.
3. Apply the frequency pgf to obtain an approximation to ϕ_A.
4. Apply the inverse FFT to create a discretized approximation to the compound cdf.

Each step involves manipulating a vector of real or complex numbers. The pgf is applied element-by-element in Step 3. For some X, such as the stable distributions, the characteristic function is known, but there is no closed-form distribution or density function. Then, it is easier to sample ϕ_X directly, replacing steps 1 and 2. See Mildenhall [2023] using FFTs to invert the characteristic function for examples.

Building on that overview, the rest of this section explains the FFT-based algorithm implemented in `Aggregate`. The discussion describes Discrete Fourier Transforms (DFTs) and their FFT implementation. We aim to demystify these concepts, show how they compute convolutions, and introduce aliasing. Understanding aliasing is critical to diagnose issues that can arise with the FFT-based algorithm, and it reveals that aliasing can be exploited in certain circumstances. In Section 4.2, we describe approaches to discretization (Step 1). In Section 4.3, we present a detailed analysis of Steps 2-4, evaluating possible errors and providing recommendations on parameter selection to maximize the accuracy of the results. The first three sections are presented without reference to the implementation, although they include code examples. Section 4.4 links the mathematical description to the `Aggregate` implementation. Finally, Section 4.5 describes the automatic validation process, an important differentiator of the `Aggregate` implementation.

4.1 Discrete Fourier Transforms and Fast Fourier Transforms

This subsection defines DFTs, explains how they compute convolutions and describes the FFT-based algorithm. To make the presentation self-contained and easy to follow, most of the formulas, straightforward in their derivation, are presented in their entirety.

Discrete Fourier Transforms

We start by defining the DFT and examining how it computes convolutions. Define an nth root of unity $\omega = \exp(-2\pi i/n)$, for integer $n \geq 1$. Euler's identity shows this is a root of unity: $\omega^n = \exp(-2\pi i) = \cos(-2\pi) + i\sin(-2\pi) = 1$. Continuing, define an $n \times n$ matrix of roots of unity

$$\mathsf{F} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w & \dots & w^{n-1} \\ 1 & w^2 & \dots & w^{2(n-1)} \\ \vdots & & & \vdots \\ 1 & w^{n-1} & \dots & w^{(n-1)^2} \end{pmatrix}.$$

The DFT of a vector $\mathsf{x} = (x_0, \dots, x_{n-1})$, typically denoted $\hat{\mathsf{x}}$, is defined simply as the matrix-vector product

$$\hat{\mathsf{x}} = \mathsf{F}\mathsf{x}.$$

Expanding the matrix multiplication shows the jth component of $\hat{\mathsf{x}}$ is

$$\hat{\mathsf{x}}_j = \sum_{k=0}^{n-1} x_k \omega^{jk}.$$

The following simple observation is very important. The formula for the sum of a geometric series shows

$$1 + \omega + \cdots + \omega^{n-1} = \frac{1-\omega^n}{1-\omega} = 0.$$

Applying the same formula to ω^j reveals the important identity

$$1 + \omega^j + \cdots + \omega^{j(n-1)} = \begin{cases} 0 & j \not\equiv 0 \pmod{n} \\ n & j \equiv 0 \pmod{n} \end{cases}.$$

The second case follows because each term in the sum equals 1. Using this identity, we can see that

$$\mathsf{F}^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & w^{-1} & \dots & w^{-(n-1)} \\ 1 & w^{-2} & \dots & w^{-2(n-1)} \\ \vdots & & & \vdots \\ 1 & w^{-(n-1)} & \dots & w^{-(n-1)^2} \end{pmatrix}$$

because the (j,k)th element of the product $\mathsf{F}\mathsf{F}^{-1}$ equals

$$\frac{1}{n}\sum_l \omega^{jl}\omega^{-lk} = \frac{1}{n}\sum_l \omega^{(j-k)l} = \begin{cases} 0 & j \neq k \\ 1 & j = k \end{cases}.$$

Moving on, we investigate how the DFT computes convolutions. Fourier transforms are like logs. Just as logarithms simplify multiplication into addition, Fourier transforms simplify convolution into multiplication. This implies that the product of DFTs should correspond to the DFT of the convolution. Let's look at the element-by-element product of the DFTs of vectors $\mathsf{x} = (x_0, \dots, x_{n-1})$ and $\mathsf{y} = (y_0, \dots, y_{n-1})$. The product of the l th elements equals

$$\left(\sum_j x_j w^{jl}\right)\left(\sum_k y_k w^{kl}\right) = \sum_{m=0}^{n-1}\left(\sum_{\substack{j,k \\ j+k\equiv m \pmod{n}}} x_j y_k\right) w^{km}$$

where the right-hand inner sum is over all j, k with $j + k = m + rn$ for some integer r. This expression shows that the product is the l th element of the DFT of the so-called wrapped or circular convolution of x and y, whose mth term is defined by the inner sum.

For example, if $n = 4$ and $m = 0$, the inner sum equals

$$x_0y_0 + x_1y_3 + x_2y_2 + x_3y_1.$$

Using arithmetic module n on the subscripts makes this look more like a convolution

$$x_0y_0 + x_1y_{-1} + x_2y_{-2} + x_3y_{-3}$$

because now the subscripts of each term sum to 0. This circular convolution differs from the expected x_0y_0 by adding tail probabilities, which "wrap-around" and reappear in the probabilities of small outcomes, a phenomenon called aliasing. The same effect makes wagon wheels appear to rotate backward in old Western movies.

The DFT convolution is an exact calculation of circular convolution, not an approximation, but we want a different calculation. The usual convolution can be obtained by padding x to the right with zeros $(x_0, x_1, x_2, x_3, 0, 0, 0, 0)$, and similarly for y. Consider the $m = 2$ component of the circular convolution, which equals

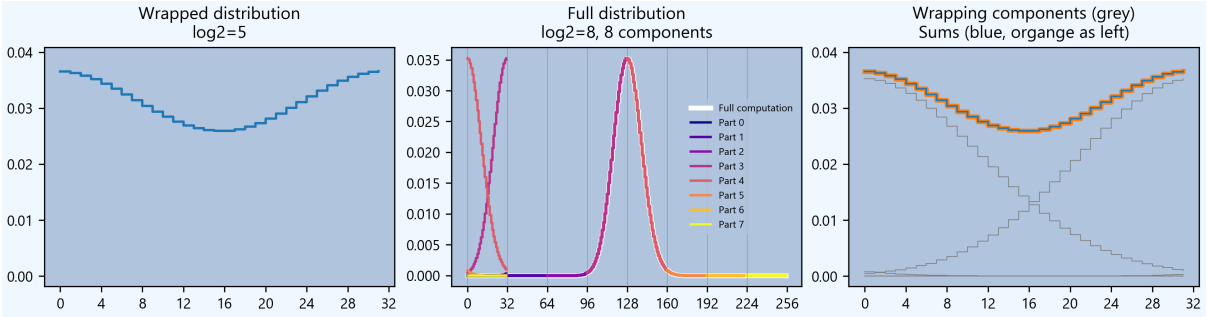
$$x_2y_0 + x_1y_1 + x_0y_2 + x_7y_3 + x_6y_4 + x_5y_5 + x_4y_6 + x_3y_7.$$

Padding reduces this to the usual convolution $x_2y_0 + x_1y_1 + x_0y_2$ for the shorter original vectors because all the other terms are zero. `Aggregate` exploits padding to obviate the impact of aliasing.

Aliasing Examples

Here are two examples of aliasing. The first shows how the FFT-based algorithm can produce apparently nonsensical results. The graphs below show an attempt to compute the pmf of a Poisson (a compound with fixed severity) with mean 128 using only $n = 32$ buckets. The severity vector $\mathsf{x} = (0, 1, 0, \dots, 0)$ has length 32 and $\mathscr{P}(z) = \exp(128(z - 1))$. The output is the inverse DFT of the vector $\mathscr{P}(\hat{\mathsf{x}})$. The result is the valley-shaped curve on the left: the aliasing effect largely averages out the underlying distribution. The middle plot shows the true distribution, centered around 128, and the vertical slices of width 32 combined to get the total. These are also shown shifted into the first slice on the left of the plot. The right plot zooms into the range 0 to 31 and shows the wrapped components that sum to the output in the first plot. The left-hand plot is a classic failure mode and a good example of how FFT methods can appear to give nonsensical results. The Python code calls a function from the `aggregate.extensions.ft` module to create the graphs.

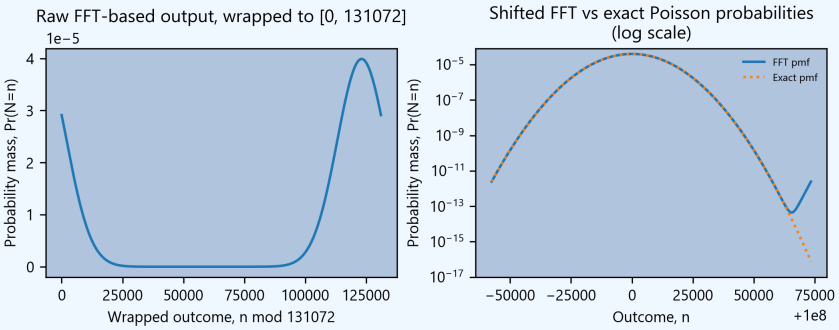
```
In [1]: import aggregate.extensions.ft as ft

In [2]: ft.fft_wrapping_illustration(ez=1, en=128, small2=5)
```



The second example shows how we can sometimes exploit aliasing to our advantage. It is often suggested that the FFT-based algorithm requires enough buckets to capture the whole range of outcomes, from zero to the distribution's right tail. In fact, it is necessary to have enough buckets only to capture the range of outcomes that occur with probability above a small threshold (say 10^{-20} or less), because adding tiny probabilities to the correct answer has no practical impact. Consider modeling a Poisson distribution with mean 100 million, $10^8 \approx 2^{27}$. It has standard deviation 10^4 and, practically, all outcomes fall in the range $10^8 \pm 5 \times 10^4$ of width $10^5 \approx 2^{17}$. The following figure applies the FFT-based algorithm to compute these probabilities with only 2^{17} buckets, less than one-thousandth of

the 2^{27} needed to capture the distribution from zero. We leverage aliasing to shift the apparently nonsensical result (left plot) to the correct range (right plot), where we see it aligns almost perfectly with the exact calculation. The error in the right tail is caused by aliasing but is too small to have any practical effect. It could be removed using 2^{18} buckets (run `ft.poisson_example(10**8, 18)`).

```
In [3]: ft.poisson_example(10**8, 17)
```



The FFT algorithm

The FFT algorithm is a surprisingly fast way to compute DFTs. It is one of the most important algorithms known to humankind and has revolutionized the practical usefulness of DFTs [Strang, 1986]. The FFT works for vectors of any length, but it is most effective for vectors of length $n = 2^g$ [Cooley and Tukey, 1965, Press *et al.*, 1992]. Computing the DFT $\mathsf{F}x$ as a matrix multiplication should take on the order of n^2 operations. The FFT exploits a much faster approach. The kth component of $\mathsf{F}x$ can be rearranged into

$$\begin{aligned}\hat{\mathsf{x}}_k &= x_0 + x_1\omega^k + x_2\omega^{2k} + x_3\omega^{3k} + x_4\omega^{4k} + \cdots \\ &= x_0 + x_2\omega^{2k} + x_4\omega^{4k} + \cdots \\ &\quad + \omega^k\left(x_1 + x_3\omega^{2k} + \cdots\right) \\ &= x_0 + x_2(\omega^2)^k + x_4(\omega^2)^{2k} + \cdots \\ &\quad + \omega^k\left(x_1 + x_3(\omega^2)^k + \cdots\right).\end{aligned}$$

Define the even and odd parts $\mathsf{x}_\mathsf{e} = (x_0, x_2, \ldots, x_{n-2})$ and $\mathsf{x}_\mathsf{o} = (x_1, x_2, \ldots, x_{n-1})$. Since ω^2 is an $n/2$th root of unity, we recognize the rearranged sum as a weighted sum of the kth elements of two smaller DFTs

$$\hat{\mathsf{x}}_k = \widehat{\mathsf{x}_\mathsf{e}}_k + \omega^k\,\widehat{\mathsf{x}_\mathsf{o}}_k$$

for $k < n/2$ and

$$\hat{\mathsf{x}}_k = \widehat{\mathsf{x}_\mathsf{e}}_l - \omega^l\,\widehat{\mathsf{x}_\mathsf{o}}_l$$

for $k = n/2 + l$, since $\omega^k = -\omega^l$ and so $\omega^{2k} = \omega^{2l}$.

Writing $O(n)$ for the fewest operations needed to compute the FFT of a vector of length n, this decomposition shows that

$$O(n) \le 2O(n/2) + 2n,$$

where the right-hand side counts the $2O(n/2)$ operations needed to compute the two shorter FFTs and then the n multiplications and n additions required to combine them. Iterating shows

$$O(n) \le 4O(n/4) + 4n \le 8O(n/8) + 6n \le \cdots \le nO(1) + 2gn.$$

Since $O(1) = 1$, this chain of inequalities shows that $O(n)$ has order at most $n\log_2(n) = ng$. For $n = 2^{20}$ (about 1 million), this a speedup factor of 50,000: the difference between 1 trillion and 20 million operations, revealing the power of the FFT algorithm.

4.2 The Discrete Representation of Distributions

We must use a discrete approximation to the exact compound cdf because most lack an analytic expression. The FFT-based algorithm and Panjer recursion both begin by replacing severity with an equally spaced discrete distribution and consequently generate an equally spaced discrete approximation to the compound cdf. Hence, the considerations discussed in this section are relevant to FFT and Panjer methods.

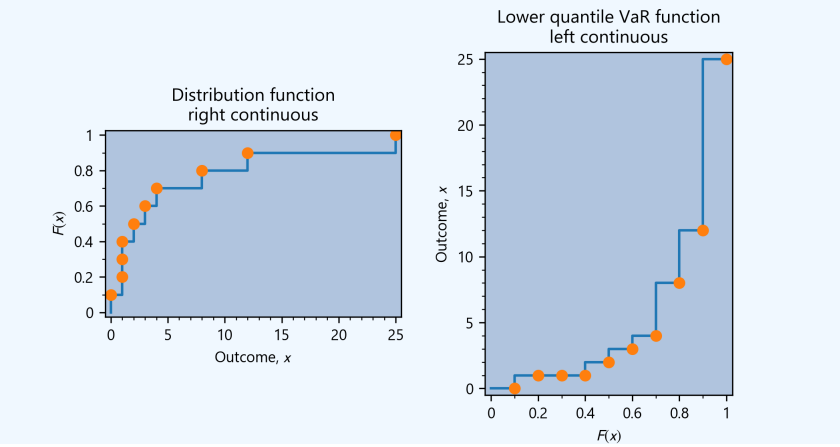
Discrete and Continuous Representations

There are two apparent ways to construct a numerical approximation to a cdf:

1. As a discrete distribution supported on a discrete set of points.
2. As a continuous distribution with a piecewise linear distribution function.

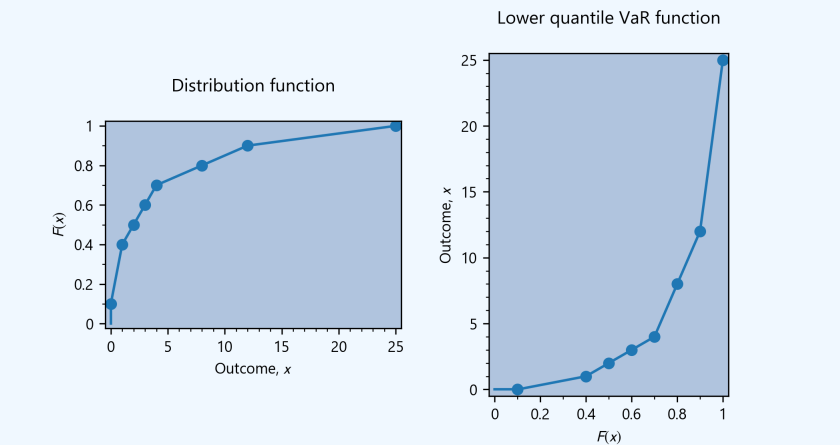
A discrete approximation produces a step-function piecewise constant cdf (left, next figure) and quantile function (right). The cdf is continuous from the right, and the (lower) quantile function is continuous from the left. The distribution does not have a density function (pdf); it only has a probability mass function (pmf).

```
In [4]: from aggregate.extensions.pir_figures import fig_4_5, fig_4_6

In [5]: fig_4_5()
```



The same two functions for a piecewise linear continuous approximation are shown below. The continuous approximation has a step-function pdf (not shown).

```
In [6]: fig_4_6()
```



The continuous approximation suggests that the compound has a continuous distribution, which is often not the case. For example, the Tweedie and all other compound Poisson distributions are mixed because they must have

a mass at zero, and a compound whose severity has a limit also has masses 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. This is one argument against continuous approximations. Another is the increased complexity. Robertson [1992] considers a quasi-FFT-based algorithm, using discrete-continuous adjustments to reflect a piecewise linear instead of a fully discrete, cdf approximation. A glance at that paper shows the adjustments greatly complicate the analysis but reveals no tangible benefits. Based on these two considerations, we use a discrete distribution approximation.

Using a discrete approximation has several implications. It means that when we compute a compound, we have a discrete approximation to its distribution function concentrated on integer multiples of a fixed bandwidth b specified by 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 compound is approximated in this way. It follows that the cdf is a step function with a jump of size p_k at kb, it is continuous from the right (it jumps up at kb), and it can be computed from the cumulative sum of $(p_0, p_1, \dots, p_{n-1})$. The approximation has rth moment given by

$$\sum_k k^r p_i b.$$

The limited expected value $\mathsf{E}[X \wedge a] = \int_0^a S(x)\,dx$ can be computed at the points $a = kb$ as b times the cumulative sum of the survival function. Finally, if the original pdf exists, it can be approximated by p_i/b.

Methods to Discretize the Severity Distribution

We need a discretized approximation to the severity distribution to apply the FFT-based algorithm. In this subsection, we discuss different ways that it can be constructed.

Let F be a distribution function of severity X and q the corresponding lower quantile function. We want to approximate F with a finite, purely discrete distribution supported at points $x_k = kb$, $k = 0, 1, \dots, m$, where b is called the bandwidth. Let's split this problem into two: first create an infinite discretization on $k = 0, 1, \dots$, and then truncate it. There are four standard 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

$$\begin{aligned} p_k &= \frac{2\mathsf{E}[X \wedge kb] - \mathsf{E}[X \wedge (k-1)b] - \mathsf{E}[X \wedge (k+1)b]}{b} \\ p_0 &= 1 - \frac{\mathsf{E}[X \wedge b]}{b}. \end{aligned}$$

   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.

Setting the first bucket to $F(b/2)$ for the rounding method (resp. $F(b)$, $F(0)$) means that any values ≤ 0 are included in the zero bucket. This behavior is useful because it allows severity to use a distribution with negative support, such as the normal or Cauchy.

Each of these methods produces a sequence $p_k \geq 0$ of probabilities that sum to 1 and that can be interpreted as the pmf and distribution function $F_b^{(d)}$ of a discrete approximation random variable $X_b^{(d)}$

$$\begin{aligned} \Pr(X_b^{(d)} = kb) &= p_k \\ F_b^{(d)}(kb) &= \sum_{i \leq k} p_i \end{aligned}$$

where superscript $d = r,\ f,\ b,\ m$ describes the discretization method and subscript b the bandwidth.

We must be clear about how the rounding method is defined and 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]

$$\begin{aligned} F_b^{(b)} &\leq F \leq F_b^{(f)} \\ F_b^{(b)} &\leq F_b^{(r)} \leq F_b^{(f)} \\ X_b^{(b)} &\geq X \geq X_b^{(f)} \\ X_b^{(b)} &\geq X_b^{(r)} \geq X_b^{(f)} \\ X_b^{(b)} &\uparrow X \text{ as } b \downarrow 0 \\ X_b^{(f)} &\downarrow X \text{ as } b \downarrow 0 \end{aligned}$$

$X_b^{(b)}$, $X_b^{(f)}$, and $X_b^{(r)}$ converge weakly (in L^1) to X as $b \downarrow 0$, and the same holds for a compound distribution with severity X. These inequalities are illustrated in *Discretization Example*.

`Aggregate` uses the rounding method by default and offers the forward and backward methods to compute explicit bounds on the distribution approximation if required. The rounding method performs well on all examples we have run. This decision is consistent with findings reported in Embrechts and Frei [2009], Klugman *et al.* [2019], Panjer [1981].

Truncation and Normalization

Next, the discrete probabilities p_k must be truncated into a finite-length vector for calculations. There are two truncation options.

1. Truncate and then normalize, dividing the truncated vector by its sum, resulting in a vector of probabilities that sums to 1.
2. Truncate without normalizing, possibly resulting in a vector of severity probabilities that sums to less than 1.

A third option, to put a mass at the maximum loss, does not produce intuitive results because the underlying distributions generally do not have such a mass.

The decision to normalize is based on the severity's tail thickness. When severity has a thin tail, there is minimal truncation error. Normalization is numerically cleaner and avoids issues with quantiles close to 1; the discrete severity probabilities should sum to 1. Policy limits in insurance applications usually bound severity, and normalization is appropriate (or has no effect). However, suppose the severity is thick-tailed, and material truncation error is unavoidable. In that case, normalization should be avoided as it can lead to unreliable results and make it difficult to interpret the estimated mean severity.

It is important to remember that the compound's right tail beyond the truncation point is understated. On the other hand, probabilities for losses below the truncation are unaffected. The bandwidth and number of buckets should be selected so that the right tail is accurate where it is being relied upon.

Approximating the Density

The compound pdf at kb can be approximated as p_k/b. This suggests another approach to discretization. Using the rounding method

$$\begin{aligned} p_k &= F((k+1/2)b) - F((k-1/2)b) \\ &= \int_{(k-1/2)b}^{(k+1)b} f(x)dx \\ &\approx f(kb)b. \end{aligned}$$

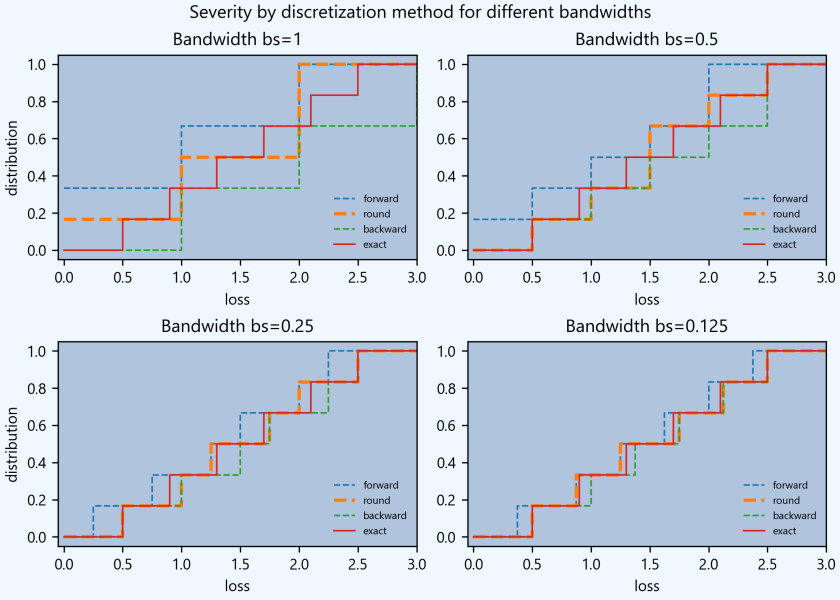
Therefore we could rescale the vector $(f(0), f(b), f(2b), \dots)$ to have sum 1. This method works well for continuous distributions but does not apply for mixed ones, e.g., when a policy limit applies.

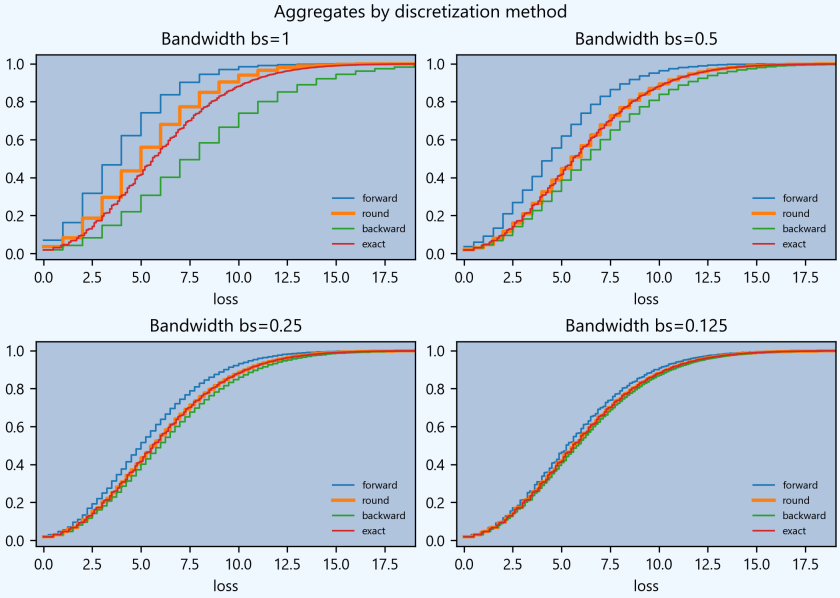
Grübel and Hermesmeier [2000] explain how to use Richardson extrapolation across varying b to obtain more accurate density estimates. While their method does improve accuracy, it is rarely necessary, given the power and speed of today's computers.

Discretization Example

This example illustrates the impact of forward, backward or rounding discretization on the resulting severity and compound distributions. The example starts from the discrete severity **dsev** `[0.5:2.5:0.4]` (i.e., $0.5, 0.9, 1.3, 1.7, 2.1, 2.5$) with equally likely outcomes. The severity has finite support, so truncation is not an issue, and it is easy to check the calculations are correct because the underlying severity is so simple. The discretized severities are shown first, using `bs=1, 1/2, 1/4, 1/8` for each method. As expected, the rounding method (orange), lies between the forward (blue) and backward (green) methods, and it is closest to the exact answer (red). The second figure shows the corresponding compound distributions, where the frequency is a mean 4 Poisson.

```
In [7]: from aggregate.extensions.figures import discretization_sev_example,␣
↪discretization_agg_example

In [8]: discretization_sev_example("0.5:2.5:0.4")

In [9]: discretization_agg_example("0.5:2.5:0.4")
```





Exponential Tilting

Grübel and Hermesmeier [1999] explain how to apply exponential tilting to the severity distribution to minimize the effect of aliasing. Tilting is also known as an exponential window [Schaller and Temnov, 2008]. Define an operator on discretized distributions by $E_\theta(\mathsf{x}) = (x_0, e^{-\theta}x_1, e^{-2\theta}x_2, \dots, e^{-(n-1)\theta}x_{n-1})$. The result is no longer a distribution, but it can still be used as one in the FFT-based algorithm. Tilting commutes with the convolution of distributions

$$E_\theta(\mathsf{x}_1) + E_\theta(\mathsf{x}_2) = E_\theta(\mathsf{x}_1 + \mathsf{x}_2)$$

because the l th term of the convolution sum equals

$$\sum_{j+k=l} e^{-j\theta}x_j e^{-k\theta}x_k = e^{-l\theta} \sum_{j+k=l} x_j x_k.$$

Therefore the tilt of a compound distribution equals the compound computed with tilted severity. Finally, we can "untilt" by applying $E_{-\theta}$, scaling the l th term up by $e^{l\theta}$.

Because of aliasing, the lth term from the FFT-based algorithm equals the expected lth term plus some wrapped terms. The expected term is multiplied by $e^{-l\theta}$ whereas the wrapped terms are all multiplied by at least $e^{-(l+n)\theta}$. Untilting multiplies them all by $e^{l\theta}$ so the wrapped terms are effectively shrunk by at least $e^{-n\theta}$ making them much smaller.

The tilt parameter should be selected to that $n\theta$ is between 20 and 36 to avoid underflow. Embrechts and Frei [2009] recommend $\theta n \le 20$. Schaller and Temnov [2008] Section 4.2 discuss other ways to select θ

Tilting is an effective way to reduce aliasing. However, once again, tilting is rarely necessary with the power of modern computers. Similar results can usually be obtained using sufficient padding. `Aggregate` offers the option to tilt, but we rarely need to use it in practice.

4.3 The `aggregate` Convolution Algorithm

Algorithm Inputs

The `Aggregate` FFT-based algorithm relies on nine inputs:

1. Ground-up severity distribution cdf and sf. See *The Severity Clause* for the set of allowable severity distributions and *The Limit Clause* and *The Reinsurance Clauses* for how the input severity can be transformed by policy limits and deductibles and per-occurrence reinsurance before frequency convolution.

2. Frequency distribution probability generating function $\mathscr{P}(z) := \mathsf{E}[z^N]$. See *The Frequency Clause* for the set of allowable frequency distributions.
3. Number of buckets $n = 2^g$, $g = \log_2(n)$. By default $g = 16$ and $n = 65536$.
4. Bandwidth, b, see *Estimating the Bandwidth*.
5. Severity calculation method: round (default), forward, or backward, see *Methods to Discretize the Severity Distribution*.
6. Discretization calculation method: survival (default), distribution, or both, see *Numerical Issues* below.
7. Normalization: true (default) or false, *Truncation and Normalization*.
8. Padding parameter, an integer $d \geq 0$ with default 1, see *Padding*.
9. Tilt parameter, a real number $\theta \geq 0$, with default 0, meaning no tilting, see *Exponential Tilting*.

The number of buckets is a user selection. On a 64-bit computer with 32GB RAM, it is practical to compute with g in the range $3 \leq g \leq 28 - d$.

Padding

Padding extends the computed severity distribution by appending zeros to increase the length to 2^{g+d}. The default $d = 1$ doubles the length of the severity vector and $d = 2$ quadruples it. Setting $d = 0$ results in no padding. Usually, $d = 1$ is sufficient but Schaller and Temnov [2008] report requiring $d = 2$ in empirical tests with very high frequency and thick tailed severity. Usually, tilting or padding is applied to manage aliasing, but not both. When both are requested, tilting is applied first, and then the result is zero-padded.

Estimating the Bandwidth

Correctly estimating the bandwidth is critical to obtaining accurate results from the FFT-based algorithm, as we saw in the aliasing examples. The bandwidth needs to be small enough to capture the shape of the severity distribution, but concurrently, it must be large enough so that the range spanned by all the buckets encompasses the shape of the compound.

`Aggregate` employs an algorithm honed through extensive trial and error, running thousands of examples. It performs well across a broad range of input assumptions. However, due to the non-local behavior of thick-tailed distributions [Mandelbrot [2013], Section 2.3.3] it is always possible to find examples where it (or any proposed algorithm) fails. Non-local behavior can lead to different moments being influenced by non-overlapping parts of the support, suggesting that different bandwidths are necessary to match different moments most accurately. Therefore, before relying on any approximation, the user should check it does not fail the built-in validation, see *Validation*.

The bandwidth is estimated from the p-percentile of a moment-matched fit to the compound. A higher value of p is used if the severity is unlimited than if it is limited. The user can also explicitly select p. Here are the details. On creation, `Aggregate` automatically computes the theoretical mean, CV, and skewness γ of the requested compound. Using those values and p, the bandwidth is estimated as follows.

1. If the CV is infinite the user must input b and an error 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, and conversely, the mean is finite if the CV exists.
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. The lognormal and gamma always have positive skewness.
4. Else if the CV is finite but skewness is infinite, fit lognormal, gamma, and normal distributions (two moments).
5. Compute the maximum policy limit L across all severity components.
6. If $L = \infty$ set $p \leftarrow \max(p, 1 - 10^{-8})$.
7. Compute b as the greatest of any fit distribution p-percentile (usually the lognormal).

8. If $L < \infty$ set $b \leftarrow \max(b, L)/n$, where n is the number of buckets, otherwise set $b \leftarrow b/n$
9. If $b \geq 1$ round to one significant digit and return.
10. Else if $b < 1$ return the smallest power of 2 greater than b, e.g., 0.2 rounds up to 0.25, 0.1 to 0.125.

Step 6 adjusts p to minimize truncation error for thick-tailed severity distributions. The last two steps ensure that b is a round number when $b \geq 1$ or an exact float when $b \leq 1$, see *Numerical Issues*.

Algorithm Steps

The algorithm steps are as follows.

1. If the frequency is identically zero, then $(1, 0, \dots)$ is returned without further calculation.
2. If the bandwidth is not specified, estimate it using *Estimating the Bandwidth*.
3. Discretize severity into a vector $\mathsf{p} = (p_0, p_1, \dots, p_{n-1})$, see *Methods to Discretize the Severity Distribution*. This step uses the discretization, severity calculation, and normalization input variables. It also accounts for policy limits and requested occurrence reinsurance.
4. If the frequency is identically one, then the discretized severity is returned without further calculation.
5. If $\theta > 0$ then tilt severity, $p_k \leftarrow p_k e^{-k\theta}$
6. Zero pad the vector p to length 2^{g+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, $\mathsf{a} := \mathsf{IFFT}(\mathsf{f})$.
10. Set $\mathsf{a} \leftarrow \mathsf{a}[0 : n]$, taking the first n entries.
11. If $\theta > 0$ then untilt, $a_k \leftarrow a_k e^{k\theta}$.
12. Apply aggregate reinsurance to a if applicable.

Grübel and Hermesmeier [1999] provides a detailed explanation of why this algorithm works. A sketch is as follows, assuming no normalization and the rounding method of discretization. As the bandwidth decreases to zero and the number of buckets increases to infinity, the discretized severity converges almost surely to the actual severity because it is bounded between the forward and backward discretizations, which decrease (increase) to severity. The compounding operator is monotonic in severity; therefore, the compound distributions with these severities also converge to the true distribution. Finally, the FFT-based algorithm differs from the actual value by, at most, the probability the actual compound distribution exceeds nb, which tends to zero.

Error Analysis

There are four sources of error in the FFT-based algorithm, each controlled by different parameters.

1. Discretization error arises from replacing the original severity distribution with a discretized approximation. It is controlled by decreasing the bandwidth.
2. Truncation error arises from truncating the discretized severity distribution at nb. It is controlled by increasing nb, which can be achieved by increasing the bandwidth or the number of buckets.
3. The FFT circular convolution calculation causes aliasing error. It is also controlled by increasing nb.
4. The discretization and FFT-based algorithms introduce numerical errors. These are usually immaterial but see *Numerical Issues*.

There is an inherent tension here: for fixed n, a smaller b is needed to minimize discretization error but a larger value to minimize truncation and aliasing error. Grübel and Hermesmeier [1999] present explicit bounds on the first three types of error.

If the input severity is discrete and bounded, the only error comes from aliasing. In particular, the algorithm can compute frequency distributions essentially exactly, as demonstrated for the Poisson in *Aliasing Examples*.

4.4 Python and `Aggregate` Implementation

This section connects the mathematical description of the FFT-based algorithm to the `Aggregate` implementation.

Python FFT Routines

Python offers a range of FFT routines. `aggregate` uses two functions from `scipy.fft` called `rfft()` and `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.

Numerical Issues

Numerical issues can trip up users used to working in spreadsheets that effectively hide floating point conniptions. It is a sad fact that computer floating point addition is not associative:

```
In [10]: x = (0.1 + 0.3) + 0.6; y = 0.1 + (0.3 + 0.6)

In [11]: print(f'{x:.25g}, {y:.25g}')
1, 0.9999999999999998889776975
```

This failure arises because 0.1 does not have an exact representation as a binary fraction, i.e., with a denominator a power of 2. As a consequence, the last element of a cumulative sum may not equal the sum computed directly, which is why the normalization option exists. The only practical solution to these problems is to use numbers with exact binary representations: replace 0.1 with 0.125 (1/8) or 0.0625 (1/16). Critically, **the bandwidth should always be a number with an exact binary representation** because multiples of the bandwidth are used to index the output compound discrete probabilities. Ignore this admonition, and you will suffer unexpected index misses and other errors – you won't be warned; you will get a headache.

The FFT routines are accurate up to machine noise, of order 10^{-16}. The noise comes from floating point issues caused by underflow and (rarely) overflow. Since the matrix F has a 1 in every row, the smallest value output by the FFT-based algorithm is the smallest x so that $1 - x \neq 1$ in the floating point implementation, which is around $2^{-53} \approx 10^{-16}$ with 64 bit floats. The noise can be positive or negative, the latter highly undesirable in probabilities. Noise 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 option `remove_fuzz` controls this behavior, and it is set `True` by default. Brisebarre *et al.* [2020] provides a thorough survey of FFT errors. They can result in large relative errors for small probabilities. See Wilson and Keich [2016] for examples and an approach to minimizing relative error.

Finally, there are numerical issues in the discretization calculation, which computes $p_k = F((k+1/2)b) - F((k-1/2)b)$. `Aggregate` supports three different calculations.

1. The distribution calculation mode 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.
2. The survival calculation mode 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 tail.
3. A combined mode attempts to make the best of both worlds, computing:

   ```
   np.maximum(np.diff(fz.cdf(adj_xs)), -np.diff(fz.sf(adj_xs)))
   ```

   It does double the work and is marginally slower.

The default is survival mode. The calculation method does not usually impact the aggregate distribution when FFTs are used because they only compute to accuracy about 10^{-16}. However, the option is helpful in creating a visually pleasing graph of severity log density.

The `update` Method

As we have discussed, two steps are required to calculate a numerical compound in `Aggregate`: specification and numerical approximation. DecL or direct API calls can specify a compound and produce a `Aggregate` object. Then, the object has a `update` method that is called to calculate the numerical approximation. Here is how the algorithm parameters listed above map to `update` options.

1. The number of buckets to use is input using `log2` to enter g. The default value is 16.
2. The bandwidth is input using `bs` ("bucket size"). If `bs=0` then then it is estimated using *Estimating the Bandwidth*.
3. Padding is input using `padding`. The default value is 1.
4. The tilt parameter is input using `tilt_vector` with a value `None` (the default) signalling no tilting.
5. Severity discretization method is selected using the `sev_calc` argument, which can take the values `round` (default), `forward`, and `backward`.
6. Discretization calculation mode is selected by `discretization_calc`, which can take the values `survival` (default), `distribution`, or `both`.
7. Normalization is controlled by `normalize`, equal to `True` (default) or `False`.
8. Remove fuzz is controlled by `remove_fuzz`, equal to `True` (default) or `False`.
9. The percentile p used to estimate the bandwidth is passed through the argument `recommend_p`. The default value is 0.99999. It is only a recommendation because the algorithm will not use $p < 1 - 10^{-8}$ for any unlimited severity.

The `build` function imported in all examples combines the specification and updating steps. It takes a DecL program and optionally any update parameters listed above as keyword arguments and returns an updated `Aggregate` object. There is much more you can do with `build`, try executing `x = build.show('Ca.Freq0(2|3|4)', verbose=True)` or `help(build)` to get started.

4.5 Validation

Validation is an important differentiating feature of the implementation. `Aggregate` automatically calculates the theoretical values of the first three moments for the gross severity and compound distributions using the well-known relationships between frequency and severity moments and compound moments [Klugman *et al.*, 2019]. The theoretical calculation uses analytic expressions for the frequency and the lognormal, gamma, exponential, and Pareto severity moments and high-accuracy numerical integration for other severities. It is performed when the object is created before any numerical approximations with `update`. After each numerical update, the approximation's moments can be compared to the theoretical moments. When they align, the user can trust that the approximation is valid. The validation workstream is entirely independent of the FFT convolution calculation, giving the user additional confidence in the results.

The validation process comprises seven tests. The tests use a relative error $\epsilon = 10^{-4}$ threshold by default, a setting that can be changed by altering the `validation_eps` global variable. The update fails validation if any of the following conditions are true.

1. The relative error in expected severity is greater than ϵ
2. The relative error in expected aggregate loss is greater than ϵ
3. The relative error in aggregate losses is more than 10 times the relative error in severity. Failing this test suggests aliasing.
4. The severity CV exists, and its relative error is greater than 10ϵ.
5. The aggregate CV exists, and its relative error is greater than 10ϵ.
6. The severity skewness exists, and its relative error is greater than 100ϵ.
7. The aggregate skewness exists, and its relative error is greater than 100ϵ.

The property `a.valid` of an `Aggregate` object `a` returns a `Validation` set of flags (a bit field) that encodes the results of each test. If no test fails, then the object returns `<Validation.NOT_UNREASONABLE: 0>`, the best outcome. Here is an example that fails validation. The compound `agg T 10 claims sev lognorm 3 poisson` is tricky to estimate (the severity CV is over 90), and with default parameters, validation returns

```
<Validation.AGG_SKEW|AGG_CV|AGG_MEAN|SEV_SKEW|SEV_CV|SEV_MEAN|NOT_UNREASONABLE: 63>
```

indicating that it fails all tests except aliasing.

If the object has reinsurance validation returns `<Validation.REINSURANCE: 128>`, because validation applies only to specifications without reinsurance, see the discussion at the end of *The Reinsurance Clauses*.

If `a` has not been updated, `a.valid` returns `<Validation.NOT_UPDATED: 256>` because there is no numerical approximation to validate.

Lastly, the method `a.explain_validation()` returns a short text description of the result. Validation information is printed automatically by `qd(a)` and when the object is printed. All of the statistics used to perform validation are contained in the `a.describe` property dataframe.

5 Examples and Workflow

`Aggregate` is a versatile tool that caters to the needs of practicing actuaries, risk management professionals, researchers, and educators alike. In this section, we detail four examples to illustrate the typical workflow. The first two examples serve the needs of actuaries and researchers by replicating exhibits from Parodi [2015] and Grübel and Hermesmeier [1999]. They illustrate various concepts such as aliasing, padding, tilting, the influence of varying bandwidths, normalization, and the choice of discretization method. They also confirm that `Aggregate` reproduces previously published results. The third example focuses on a more complex pricing problem for actuaries and is based on an intricate spliced mixed severity distribution from Albrecher *et al.* [2017]. The fourth is tailored to meet the pedagogical needs of teachers and students. Readers should actively engage with the code to get the most out of this section.

The examples illustrate the recommended specify-update-validate-adjust-use (SUVA-use) workflow:

- Specify the gross compound using DecL.
- Update the numerical approximation using the `update` method (performed automatically for objects created using DecL and `build`).
- Validate the results are "not unreasonable" by reviewing the diagnostics; if necessary, adjust update parameters and re-run `update`.
- Adjust the specification for reinsurance and update using the same parameters.
- Use the output.

The SUVA-use workflow leverages the built-in validation on objects without reinsurance and helps ensure the validity of net and ceded distributions.

5.1 Discretization Method and Varying Bandwidth

The first example reproduces an exhibit from Parodi [2015] that explores quantiles of a compound with mean 3 Poisson frequency, and lognormal severity with $\mu = 10$ and $\sigma = 1$.

Parodi uses 65,536 buckets, the `Aggregate` default, and recomputes the compound using bandwidths of 10, 50, 100, and 1000. He then extracts various percentiles.

Translating the Poisson-lognormal compound into DecL is straightforward: σ is the shape parameter, and $\mu = 10$ translates into a scale factor $\exp(10)$. (A `scipy.stats.lognorm` object with shape σ is lognormal with $\mu = 0$.) Running the model with `normalize=False`, because severity is unlimited and thick-tailed *Truncation and Normalization*, and taking defaults for other update parameters produces an approximation that is not unreasonable. `Aggregate` selects a bandwidth of 70.

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

In [2]: a = build('agg Parodi 3 claims sev exp(10) * lognorm 1 poisson'
   ...:           , normalize=False)
   ...:

In [3]: 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        36316       36315 -1.2535e-06  1.3108    1.3107   6.1849      6.1431
Agg   1.0895e+05 1.0895e+05 -1.4149e-06 0.95189   0.95181   2.5875      2.5743
log2 = 16, bandwidth = 70, validation: not unreasonable.

In [4]: print(f'mean={a.agg_m:,.0f}, sd={a.agg_sd:,.0f}')
mean=108,947, sd=103,705
```

The tabular output above shows the analytic expected value `E[X]`, `CV(X)` and skewness `Skew(X)` for the frequency, severity, and aggregate distributions. The columns prefixed `Est` show the corresponding statistics computed using the FFT-based calculation. The mean matches with a relative error of 1.4×10^{-6}, within the validation tolerance of 10^{-4} (resp. within 10^{-3}, 10^{-2} for cv and skewness; the latter errors not shown). Users should always review this table before using the numerical output.

Updating the object with `normalize=True` shows the impact of this setting. The discretized severity CV and skewness statistics are closer to their actual values without normalization.

Continuing, we can recalculate the `Aggregate` object `a` to match Parodi's calculations, using the `update` method to try different parameter settings. Parodi discretizes severity by re-scaling the density evaluated on multiples of the bucket size. This method close to the `sev_calc='backward'`, see *Methods to Discretize the Severity Distribution*. He does not discuss padding, so we set `padding=0`. Assembling the parts and running with some basic Python code produces the following table.

```
In [5]: import pandas as pd

In [6]: means = ['Mean', a.est_m]; sds = ['SD', a.est_sd]

In [7]: p = np.array([.5, .75, .8, .9,  .95, .98, .99,  .995, .998, .999, .9999, .
↪99999])

In [8]: df = pd.DataFrame({'p': p, 'Aggregate': a.q(p)})

In [9]: for bs in [100, 50, 10, 1000, '1000']:
   ...:     if type(bs) == str:
   ...:         sev_calc = 'discrete'; bs = float(bs); cn = 'h=1000*';
   ...:     else:
   ...:         sev_calc = 'backward'; bs = float(bs); cn = f'h={bs:.0f}';
   ...:     a.update(bs=bs, log2=16, padding=0, sev_calc=sev_calc)
   ...:     df[cn] = a.q(p)
   ...:     means.append(a.est_m)
   ...:     sds.append(a.est_sd)
   ...:

In [10]: df.loc['Mean', :] = means

In [11]: df.loc['Error on Mean', :] = ['Error on Mean'] + [f'{m / a.agg_m - 1:.2%}'
   ....:                               for m in means[1:]]
   ....:

In [12]: df.loc['SD', :] = sds

In [13]: df.loc['Error on SD', :] = ['Error on SD'] + [f'{s / a.agg_sd - 1:.2%}'
```

(continues on next page)

(continued from previous page)

```
   ....:                                 for s in sds[1:]]
   ....:

In [14]: df['p'] = [i if type(i) == str else (f'{i:.1%}' if i <= 0.999 else f'{i:.3
↪%}') for i in df.p]

In [15]: qd(df.set_index('p'), ff=lambda x: f'{x:10,.0f}')

               Aggregate      h=100       h=50       h=10     h=1000    h=1000*
p
50.0%             82,320     82,500     82,400     82,090     84,000     82,000
75.0%            148,820    149,100    148,950    148,140    151,000    149,000
80.0%            169,610    169,800    169,750    168,680    172,000    170,000
90.0%            234,570    234,800    234,700    232,390    237,000    235,000
95.0%            302,050    302,300    302,150    297,160    304,000    302,000
98.0%            398,020    398,300    398,150    384,660    401,000    398,000
99.0%            478,030    478,300    478,100    450,200    481,000    478,000
99.5%            566,230    566,500    566,350    511,030    569,000    566,000
99.8%            699,160    699,400    699,200    575,820    702,000    699,000
99.9%            815,010    815,300    814,950    609,190    818,000    815,000
99.990%        1,329,510  1,329,700  1,326,800    649,890  1,332,000  1,330,000
99.999%        2,120,370  2,119,700  2,079,650    654,800  2,123,000  2,120,000
Mean             108,946    109,096    109,018    107,019    110,446    108,947
Error on Mean     -0.00%      0.14%      0.07%     -1.77%      1.38%      0.00%
SD               103,697    103,752    103,652     96,020    104,234    103,706
Error on SD       -0.01%      0.04%     -0.05%     -7.41%      0.51%      0.00%
```

The last table agrees closely with Parodi's book, reproduced below. Interestingly, almost all the error across bandwidths > 10 is caused by the way Parodi discretizes. The last column `h=1000*` shows the result using the `rounding` method: it is much closer to the first column. In particular, the mean is almost exact. When `h=10` there is too much truncation error to obtain accurate results.

FFT Gross Loss Model Results for Different Discretions Steps

Percentile	$h = 100$	$h = 50$	$h = 10$	$h = 1000$	Exact
50.0%	82,500	82,400	82,160	84,000	
75.0%	149,100	148,950	148,380	151,000	
80.0%	169,800	169,750	169,010	172,000	
90.0%	234,800	234,700	233,200	237,000	
95.0%	302,300	302,150	298,970	304,000	
98.0%	398,300	398,200	389,590	401,000	
99.0%	478,300	478,150	460,080	481,000	
99.5%	566,500	566,350	529,340	569,000	
99.8%	699,400	699,250	610,540	702,000	
99.9%	815,300	815,100	655,360	818,000	
Mean	109,096	109,017	106,872	110,446	**108,947**
Error on mean	0.14%	0.06%	−1.90%	1.38%	
SD	103,752	103,652	96,017	104,234	**103,705**
Error on SD	0.05%	−0.05%	−7.41%	0.51%	

FIGURE 17.5
Aggregate loss model calculated with FFT with different discretisation steps, but with the same number of discretisation points (65,536). The last column shows the theoretical value of the mean and the standard deviation, as calculated by Equations 6.3 and 6.4 (as to the percentiles, the exact value is unknown).

Fig. 1: Figure 17.5 from Parodi [2015].

This example highlights the importance of the discretization method, the superiority of the discrete rounding method, and the need to select the bandwidth carefully.

5.2 Aliasing, Tilting, and Padding

The second example reproduces a table from Grübel and Hermesmeier [1999] based on a compound with mean 20 Poisson frequency and a Levy stable severity with $\alpha = 1/2$. The Levy distribution is a zero parameter distribution in `scipy.stats`.

The mean of the Levy distribution does not exist, implying that `Aggregate` cannot estimate the bandwidth, so the user must supply it to avoid an error. The example takes `bs=1`. In this case, the diagnostics are irrelevant, and the usual SUVA-use workflow is adjusted. To validate, we use the stable property to compute the compound probabilities exactly. Being stable with index $\alpha = 1/2$ means that

$$X_1 + \cdots + X_n \sim n^2 X.$$

This identity and conditional probability are used to compute the compound probability that $x - 1/2 < X \le x + 1/2$.

The `aggregate.extensions.figures` function `gh_figure` reproduces Table 1 from the original paper. It starts by creating **`agg`** `GH.Eg 20` **`claims sev`** `levy poisson` with `update=False`. The first table shows the diagnostics dataframe before updating. Then `update` is applied with `bs=1` and various other input parameters.

```
In [16]: from aggregate.extensions.figures import gh_example

In [17]: gh_example(20)

      E[X]   CV(X) Skew(X)
X
Freq    20 0.22361 0.22361
Sev    inf
Agg    inf
log2 = 0, bandwidth = na, validation: n/a, not updated.

               True   Agg pad=2      Tilt 0  Tilt 0.00098  Tilt 0.0049  Tilt 0.024
x
     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   1.155e-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
```

The last table is identical to that shown in the paper, reproduced below. The Panjer column, computed using Panjer recursion in the original paper, is replicated by `Aggregate` using `log2=16`, `padding=2`, and `bs=1`. The remaining models use `bs=1` with `log2=10`, no padding, and varying amounts of tilting as shown in the column headings.

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

Fig. 2: Table 1 from Grübel and Hermesmeier [1999].

This example confirms that padding is an effective way to combat aliasing. Padding is much simpler to implement than tilting and is the `Aggregate` default. The example also shows the FFT-based algorithm matches the exact probabilities computed analytically. Running with `normalize=True` shows the impact of normalizing a thick-tailed severity. Mildenhall [2023] presents other similar examples from Embrechts and Frei [2009], Schaller and Temnov [2008], and Shevchenko [2010].

5.3 A Complex Pricing Problem

The third example builds a more complex model, similar to one an actuary might use to exposure rate a per-occurrence or aggregate excess reinsurance treaty. The analysis is from the cedent's perspective, focusing on the distribution of net outcomes. Following the SUVA-use workflow, the example is built up in stages, starting with the gross distribution. The underlying book is motor third-party liability, and we use realistic parameters determined by Albrecher *et al.* [2017] for MTPL Company A, Section 6.6.

Albrecher models frequency using a Poisson distribution with an annual expected claim count of 55.27. Severity is a splice mixture of a gamma (Erlang) mixture and a single parameter Pareto (ibid. p.110). There are two gamma components with shapes 1 and 4, common scale 63410, and weights 0.155 and 0.845, respectively. The Pareto distribution is spliced in for claims above 500,000 and has a shape parameter equal to $1/0.506 = 1.976$ (mean but no variance). The gamma mixture has weight 0.777. The next code block creates a named severity `MTPL.A` with these parameters that we can use in subsequent calculations. It uses Python `f`-strings to substitute values into the DecL program.

```
In [18]: from aggregate import build, qd, pprint_ex

In [19]: import pandas as pd

In [20]: pi = 0.777; alpha = np.array([0.155, 0.845]); r = np.array([1, 4])

In [21]: scale = 63410.; pareto_alpha = 1 / 0.506; t = 500000.

In [22]: wts = np.hstack((alpha * pi, 1-pi))

In [23]: shape = np.hstack((r, pareto_alpha))

In [24]: s = build(f'sev MTPL.A [{scale} {scale} {t}] * '
   ....:           f'[gamma gamma pareto] {shape}  '
   ....:           f'wts {wts} splice [0 0 {t}] [{t} {t} inf] ')
   ....:

In [25]: print(pprint_ex(s.program, split=60))
sev MTPL.A [63410.0 63410.0 500000.0] * [gamma gamma pareto] [1. 4. 1.97628458]
  wts [0.120435 0.656565 0.223 ] splice [0 0 500000.0] [500000.0 500000.0 inf]
```

Next, we build the gross compound distribution, using the DecL `sev.MTPL.A` to recall the severity created in the last step. Since the Pareto component has no variance, we must input a bandwidth, here 5000. We also increase the default number of buckets, using `log2=19`. Finally, we set `normalization=False` since severity is uncapped and thick-tailed. We show the validation dataframe and report quantiles that closely match those shown in Albrecher Table 6.1.

```
In [26]: gross = build('agg Gross 55.27 claims sev.MTPL.A poisson'
   ....:               , bs=5000, log2=19, normalize=False)
   ....:

In [27]: qd(gross)

           E[X]    Est E[X]    Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq      55.27                          0.13451           0.13451
Sev  3.8928e+05 3.8925e+05 -7.5109e-05     inf    2.5796              517.08
Agg  2.1515e+07 2.1514e+07 -7.5663e-05     inf   0.37196              56.117
```

(continues on next page)

(continued from previous page)

```
log2 = 19, bandwidth = 5000, validation: not unreasonable.

In [28]: df = pd.DataFrame({'p': [0.95, 0.99, 0.995]})

In [29]: df['Sev VaR'] = gross.q_sev(df.p)

In [30]: df['Alb sev'] = [1065451., 2405538., 3416123.]

In [31]: df['Agg VaR'] = gross.q(df.p)

In [32]: df['Alb agg'] = [31100000., 41420000., 48760000.]

In [33]: qd(df,
   ....:     ff=lambda x: f'{x:.1%}' if x < 1 else f'{x:12,.0f}', index=False)
   ....:

    p      Sev VaR      Alb sev      Agg VaR      Alb agg
95.0%    1,065,000    1,065,451   31,160,000   31,100,000
99.0%    2,405,000    2,405,538   41,505,000   41,420,000
99.5%    3,415,000    3,416,123   48,915,000   48,760,000
```

The property `gross.statistics` returns an expanded table of theoretic frequency, severity, and compound statistics by severity mixture component (not displayed).

In the SUVA-use workflow, we have fixed the specification of the gross portfolio and determined appropriate update parameters. Moving on, we adjust the gross by adding an occurrence reinsurance tower. The DecL recalls the gross specification using `agg.Gross` and then appends the reinsurance, ensuring consistency between gross and net. The tower has two layers, a 50% share (placement) of 500,000 xs 500,000 and a 100% placement of 1M xs 1M.

```
In [34]: net = build('agg Net agg.Gross occurrence net of '
   ....:             '50% so 500000 xs 500000 and 1000000 xs 1000000 '
   ....:             , bs=5000, log2=19, normalize=False)
   ....:

In [35]: qd(net)

           E[X]    Est E[X] Err E[X]   CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq      55.27                      0.13451           0.13451
Sev  3.8928e+05 3.3283e+05   -0.145     inf    2.7731              661.81
Agg  2.1515e+07 1.8396e+07   -0.145     inf   0.39633              73.833
log2 = 19, bandwidth = 5000, validation: n/a, reinsurance.

In [36]: df['Net Sev VaR'] = net.q_sev(df.p)

In [37]: df['Net Agg VaR'] = net.q(df.p)

In [38]: df['Occ Chg'] = df['Net Agg VaR'] / df['Agg VaR'] - 1

In [39]: qd(df.drop(columns=['Alb sev', 'Alb agg']), ff=lambda x: f'{x:.1%}' if x
↪< 1 else f'{x:12,.0f}', index=False)

    p      Sev VaR      Agg VaR  Net Sev VaR  Net Agg VaR  Occ Chg
95.0%    1,065,000   31,160,000      750,000   25,860,000   -17.0%
99.0%    2,405,000   41,505,000    1,155,000   36,195,000   -12.8%
99.5%    3,415,000   48,915,000    2,165,000   43,835,000   -10.4%
```

The reinsurance has the anticipated impact on losses, lowering the tail quantiles by 10-17%. Finally, we add an aggregate excess of loss program, covering from the 95th to the 99th percentiles of net losses. We start with the net program and append the new program, passing in the limit and attachment using the quantile function on the net. The combined program lowers the tail quantiles by between 17% and 38%.

```
In [40]: net_agg = build('agg Net-Net agg.Net aggregate net of '
   ....:                 f'{net.q(0.99) - net.q(0.95)} xs {net.q(0.95)} '
   ....:                  , bs=5000, log2=19, normalize=False)
   ....:

In [41]: qd(net)

            E[X]    Est E[X] Err E[X]    CV(X) Est CV(X) Skew(X) Est Skew(X)
X
Freq       55.27                       0.13451           0.13451
Sev   3.8928e+05 3.3283e+05   -0.145      inf    2.7731              661.81
Agg   2.1515e+07 1.8396e+07   -0.145      inf   0.39633              73.833
log2 = 19, bandwidth = 5000, validation: n/a, reinsurance.

In [42]: df['Net-net Agg'] = net_agg.q(df.p)

In [43]: df['Agg Chg'] = df['Net-net Agg'] / df['Agg VaR'] - 1

In [44]: qd(df.drop(columns=['Alb sev', 'Alb agg', 'Net Sev VaR',  'Occ Chg']),␣
↪ff=lambda x: f'{x:.1%}' if x < 1 else f'{x:12,.0f}', index=False)

    p       Sev VaR       Agg VaR  Net Agg VaR  Net-net Agg  Agg Chg
95.0%     1,065,000    31,160,000   25,860,000   25,860,000   -17.0%
99.0%     2,405,000    41,505,000   36,195,000   25,860,000   -37.7%
99.5%     3,415,000    48,915,000   43,835,000   33,500,000   -31.5%

In [45]: print(net_agg.reinsurance_description())
Net of 50.00% share of 500,000.00 xs 500,000.00 and 100.00% share of
1,000,000.00 xs 1,000,000.00 per occurrence then net of 100.00% share
of 10,335,000.00 xs 25,860,000.00 in the aggregate.
```

This analysis is from the cedent's perspective, focusing on the distribution of net outcomes. It can be switched to the reinsurer's perspective by replacing **net of** with **ceded to** to obtain the distribution of ceded losses for each treaty. One caveat: it is impossible to obtain the total occurrence plus aggregate cession in one step using `Aggregate`.

By adjusting the occurrence and aggregate layers, it is possible to compute Table L and M charges [Fisher *et al.*, 2017] using this template. The template can be extended to incorporate a limits profile and more complex reinsurance arrangements, such as sliding scale or profit commissions, loss corridors, or swing rating [Bear and Nemlick, 1990, Clark, 2014].

This example uses a realistic, three-way conditional mixture to model a multi-layer occurrence and aggregate excess of loss program. It illustrates the SUVA-use workflow and shows how DecL's ability to store and recall compounds provides a succinct and reliable way to specify complex structures.

5.4 A Discrete Compound for Educators and Students

The fourth and final example uses `Aggregate` as an educational tool by setting up and solving a textbook problem about a compound with discrete frequency and severity.

Question. You are told that frequency N can equal 1, 2, or 3, with probabilities 1/2, 1/4, and 1/4, and that severity X can equal 1, 2, or 4, with probabilities 5/8, 1/4, and 1/8. Model the compound distribution $A = X_1 + \cdots + X_N$ using the collective risk model and answer the following questions.

1. What are the expected value, CV, and skewness of N, X, and A?
2. What possible values can A take? What are the probabilities of each?
3. Plot the pmf, cdf, and the outcome Lee diagram for severity and the compound distribution.

Solution: Use the `dfreq` and **dsev** DecL keywords to specify the frequency (exposure) and severity by entering vectors of outcomes and probabilities. This alternative syntax is very convenient for this type of problem. The

FFT-based algorithm is exact for models with discrete severity and bounded frequency because padding removes aliasing, and there is no discretization error. Thus, the validation dataframe answers question 1.

```
In [46]: from aggregate import build, qd

In [47]: ex = build('agg Discrete.Eg dfreq [1 2 3] [1/2 1/4 1/4] dsev  [1 2 4] [5/
↪8 1/4 1/8]')

In [48]: qd(ex)

       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.
```

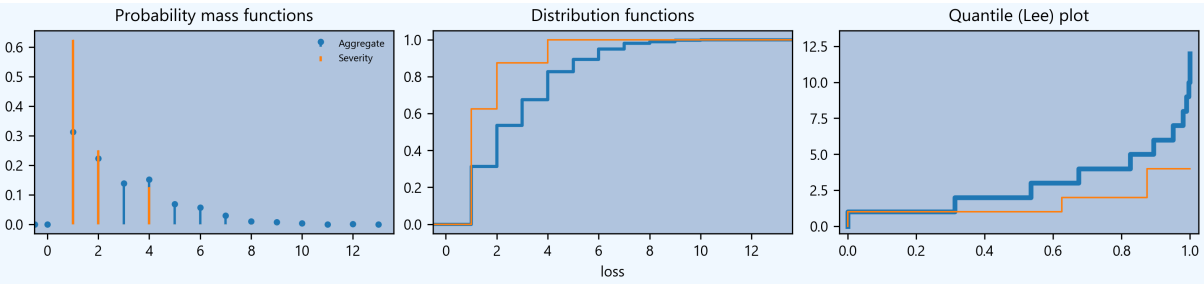
The aggregate pmf is available in the dataframe `ex.density_df`. Here are the pmf, cdf, and sf evaluated for all possible outcomes, answering question 2. The index is adjusted to an `int` to print nicely.

```
In [49]: bit = ex.density_df; bit.index = bit.index.astype(int)

In [50]: def f(x):
   ....:     ir = np.round(x, 13).as_integer_ratio()
   ....:     return f'{int(x)}' if x in [0, 1] else f'  {ir[0]}/{ir[1]}'
   ....:

In [51]: qd(bit.query('p_total > 0')[['p_total', 'F', 'S']], ff=f)

         p_total           F          S
loss
1           5/16        5/16      11/16
2          57/256     137/256    119/256
3        285/2048   1381/2048   667/2048
4        155/1024   1691/2048   357/2048
5          35/512   1831/2048   217/2048
6        115/2048    973/1024    51/1024
7          15/512   1003/1024    21/1024
8           5/512   1013/1024    11/1024
9         15/2048   2041/2048     7/2048
10         3/1024   2047/2048     1/2048
12         1/2048           1          0
```

The possible outcomes range from 1 (frequency 1, outcome 1) to 12 (frequency 3, all outcomes 4). It is easy to check that the reported probabilities are correct, and a moment's thought confirms that obtaining an outcome of 11 is impossible.

Finally, the graphs requested in question 3 are produced by `ex.plot()` using data in the `ex.density_df` dataframe. They automatically use settings appropriate to a discrete distribution.

```
In [52]: ex.plot()
```



This example shows how `Aggregate` can be used to set up and solve simple classroom problems about compound

distributions and illustrates DecL's very concise syntax for discrete distributions.

6 Conclusion

The collective risk model, a foundation of actuarial science, relies on frequency-severity compound distributions. `Aggregate` brings these models to life by approximating compound distributions using a fast, accurate, and flexible FFT-based algorithm. It simplifies the use of FFTs by providing default values for the bandwidth and other crucial parameters and includes a validation process to bolster user confidence. DecL, a domain-specific language introduced by `Aggregate`, enables users to specify compound distributions using familiar insurance and actuarial terms. Distributions are parameterized using the mean, CV, and other user-relevant quantities. To our knowledge, `Aggregate` is the first package to speak the user's language in this way.

Practicing actuaries, researchers, teachers, and students can all benefit from using `Aggregate`. Its speed, accuracy, and flexibility make it an ideal tool for actuaries working in large account, reinsurance, excess of loss, or property pricing, risk management, and catastrophe risk management, or any other application requiring the entire distribution of potential outcomes. Researchers can use it to observe subtle distributional properties, test hypotheses, and create edifying examples swiftly and simply. Finally, `Aggregate` is a workbench on which teachers and students can set up and solve a wide variety of textbook problems about frequency, severity, and compound distributions. It helps students see the underlying theory instead of a morass of mechanical computations.

The `Aggregate` class is part of the larger `aggregate` Python package that aims to broaden the appeal of FFT-based methods. The time is ripe for this contribution. Today, the actuarial profession is more willing to use open-source software, spurred on by the successful adoption of R and Python in machine learning and predictive modeling. `Aggregate` can help FFT-based algorithms become the mainstream methods they deserve to be.

References

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[ABT17] Hansjörg Albrecher, Jan Beirlant, and Jozef L Teugels. *Reinsurance: Actuarial and Statistical Aspects*. John Wiley & Sons, 2017.

[AA10] Soren Asmussen and Hansjorg Albrecher. *Ruin probabilities*. Volume 14. World scientific, 2010.

[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.

[Bea22] David Beazely. Sly (sly lex-yacc). 2022. URL: https://github.com/dabeaz/sly.

[Ber81] Jürgen Bertram. Numerische berechnung von gesamtschadenverteilungen. *Blätter der DGVFM*, 15(2):175–194, 1981.

[Bil86] Patrick Billingsley. *Probability and Measure*. J. Wiley and Sons, second edition, 1986.

[Boh69] Harald Bohman. The numerical integration of the Fourier inversion formula for distribution functions. *Scandinavian Actuarial Journal*, 1969(3):52–62, 1969. doi:10.1080/03461238.1969.10404608.

[Boh74] Harald Bohman. Fourier inversion-distribution functions-long tails. *Scandinavian Actuarial Journal*, 1974(1):43–45, 1974. doi:10.1080/03461238.1974.10408660.

[BJM+20] Nicolas Brisebarre, Mioara Joldeş, Jean Michel Muller, Ana Maria Naneş, and Joris Picot. Error Analysis of Some Operations Involved in the Cooley-Tukey Fast Fourier Transform. *ACM Transactions on Mathematical Software*, 2020. doi:10.1145/3368619.

[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.

[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.

[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.

[CT65] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965. doi:10.2307/2003354.

[CT21] Dan Corro and Yen-chieh Tseng. NCCI's 2014 Excess Loss Factors. *Variance*, 14(1):1–28, 2021.

[DPP94] C D Daykin, T Pentikainen, and M Pesonen. *Practical Risk Theory for Actuaries*. Chapman and Hall, 1994.

[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.

[DGP08] Christophe Dutang, Vincent Goulet, and Mathieu Pigeon. Actuar: An R package for Actuarial Science. *Journal of Statistical Software*, 25(7):1–37, 2008. doi:10.18637/jss.v025.i07.

[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.

[EKluppelbergM13] Paul Embrechts, Claudia Klüppelberg, and Thomas Mikosch. *Modelling extremal events: for insurance and finance*. Volume 33. Springer Science & Business Media, 2013.

[Fel71] William Feller. *An Introduction to Probability Theory and its Applications, Volume 2*. J. Wiley and Sons, second edition, 1971. ISBN 0471257095.

[FMPP17] Ginda Kaplan Fisher, Lawrence McTaggart, Jill Petker, and Rebecca Pettingell. Individual Risk Rating. Technical Report April, Casualty Actuarial Science, 2017.

[Fre18] Edward Frees. *Loss Data Analytics*. https://openacttexts.github.io/Loss-Data-Analytics/, 2018. URL: http://arxiv.org/abs/1808.06718, arXiv:1808.06718.

[GDVH84] Marc Goovaerts, Florent Etienne De Vylder, and Jean Haezendonck. *Insurance premiums*. North-Holland, 1984.

[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.

[Hip06] Christian Hipp. Speedy convolution algorithms and Panjer recursions for phase-type distributions. *Insurance: Mathematics and Economics*, 38(1):176–188, 2006. doi:10.1016/j.insmatheco.2005.08.009.

[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.

[HMS83] Gur Huberman, David Mayers, and Clifford W. Smith. Optimal Insurance Policy Indemnity Schedules. *The Bell Journal of Economics*, 14(2):415, 1983. doi:10.2307/3003643.

[Hun07] J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[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.

[JKK05] Norman L Johnson, Samuel Kotz, and Adrienne W Kemp. *Univariate discrete distributions*. John Wiley & Sons, 2005.

[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.

[Kyp14] Andreas E Kyprianou. *Fluctuations of Lévy processes with applications: Introductory Lectures*. Springer Science & Business Media, 2014.

[Korner22] Thomas William Körner. *Fourier analysis*. Cambridge university press, 2022.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 1992.

[Loe55] Michel Loeve. *Probability Theory*. D. Van Nostrand Company, 1955. ISBN 9780486814889. doi:10.1137/1006078.

[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.

[MAKL95] Paul Malliavin, Hélène Airault, Leslie Kay, and Gérard Letac. *Integration and probability*. Volume 157. Springer Science & Business Media, 1995.

[Man13] Benoit B Mandelbrot. *Fractals and scaling in finance: Discontinuity, concentration, risk. Selecta volume E*. Springer Science & Business Media, 2013.

[McK14] Henry McKean. *Probability: the classical limit theorems*. Cambridge University Press, 2014.

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference*, 1(Scipy):56–61, 2010. doi:10.25080/majora-92bf1922-00a.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005. doi:10.1115/IPC2010-31470.

[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.

[Mil23] Stephen J. Mildenhall. Aggregate documentation. 2023. URL: https://aggregate.readthedocs.io/_/downloads/en/latest/pdf/.

[MM22] Stephen J. Mildenhall and John A. Major. *Pricing Insurance Risk: Theory and Practice*. John Wiley & Sons, Inc., 2022.

[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.

[pdt20] The pandas development team. Pandas-dev/pandas: pandas. February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134.

[Pan81] Harry H Panjer. Recursive evaluation of a family of compound distributions. *ASTIN Bulletin*, 12(1):22–26, 1981. doi:10.1017/S0515036100006796.

[Par15] Pietro Parodi. *Pricing in General Insurance*. CRC Press, 2015. ISBN 9781466581487.

[PLM22] Gabriele Pittarello, Edoardo Luini, and Manfred Marvin Marchione. GEMAct: a Python package for non-life (re)insurance modelling. *Annals of Actuarial Science*, 0:4–11, 2022.

[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.

[Rob92] John P. Robertson. The computation of aggregate loss distributions. *Proceedings of the Casualty Actuarial Society*, 79(150):57–133, 1992.

[RU02] R. Tyrrell Rockafellar and Stanislav Uryasev. Conditional value-at-risk for general loss distributions. *Journal of Banking & Finance*, 26(7):1443–1471, 2002.

[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.

[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.

[Str86] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[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.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[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.

[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.

[Zhu11] Li Zhu. Introduction to Increased Limit Factors. In *CAS Ratemaking Seminar*, 1–31. 2011.

[vCerny04] Aleš Černý. Introduction to Fast Fourier Transform in Finance. *Journal of Derivatives*, 12(1):73–88, 2004.