Annals of Actuarial Science (2024), 1–40
doi:10.1017/S1748499524000216

ACTUARIAL SOFTWARE

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

Stephen Mildenhall

Independent Scholar, London, UK
Email: steve@convexrisk.com

(Received 31 July 2023; revised 30 May 2024; accepted 04 June 2024)

Abstract

`Aggregate` implements an efficient fast Fourier transform (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 nonprogrammers 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; FFT; Fourier transform; Python; risk aggregation

1. Introduction

1.1. 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, 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 & Haezendonck, 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 problems to solve when computing a compound: how to estimate its distribution probabilities quickly and accurately and how to specify it flexibly yet succinctly.

Estimating compound distributions is computationally difficult. Here, a 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, p. 201) says:

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

Implementing an FFT-based method 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. FFT-based methods can fail unexpectedly, giving apparently nonsensical results, and they are 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 & Drake, 2009) implementation. To achieve this aim, it introduces two innovations.

- `Aggregate` validates the FFT-based algorithm output by comparing the first three moments of its approximations against their theoretical values (when they exist) 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 Section 4.7 for the details.
- By offering reasonable default values for all parameters, `Aggregate` clears a significant hurdle to the widespread adoption of FFT-based methods, what Parodi (2015, p. 245) 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 Section 4.2 for details. Users can accept the defaults or override them based on their specific needs.

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 multilayer reinsurance structures. `Aggregate` addresses this challenge through its third innovation: the domain-specific language DecL. DecL acts as a mid-point between an application programmer's interface (API) and a GUI-based user interface. It provides a user-friendly way to specify intricate, real-world distributions, employing a straightforward and human-readable syntax. DecL's specification centers on a vocabulary and quantities familiar to actuaries, like the mean loss ("loss pick") or severity CV ("volatility"). DecL is described in Section 3.

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

- Access to over 100 continuous severity distributions, and a flexible discrete severity distribution, see Section 3.8.
- Access to a broad selection of frequency distributions, including mixed Poisson, zero modified, and zero truncated distributions, see Section 3.9.
- The ability to apply policy limits and attachments and specify a full limits profile, perform excess of loss exposure rating, overlay complex per-occurrence and aggregate reinsurance towers with coparticipations, and value aggregate features such as Table L and M charges, sliding scale and profit commissions, aggregate deductibles and limits, and price swing rated programs. See Sections 3.7 and 3.11.

- Built-in cumulative distribution `cdf`, survival `sf`, and probability density and mass `pdf`, `pmf` functions, as well as quantiles (value at risk) `q`, and TVaR `tvar`. The underlying computations explicitly allow for the discrete approximation used (Rockafellar & Uryasev, 2002). These statistics are important in risk management and catastrophe risk evaluation.
- A library of over 150 illustrative examples. Each can be created by name, or they can be built in a batch.
- Built-in calculation of the probability of eventual ruin as a function of the premium rate and starting surplus for a compound Poisson process using the Pollaczeck–Khinchine (PK) formula (Embrechts et al., 2013, Section 1.3).
- The ability to compute technical (risk-loaded) premiums using spectral risk measures at different capital standards (Mildenhall & Major, 2022).
- Integration with standard Python packages provides access to their powerful scientific computing, data manipulation, and visualization methods. For example, severity distributions use `scipy.stats` (Virtanen et al., 2020) continuous random variable pdfs and cdfs adjusting for limits and attachments, and output is delivered using `pandas` `DataFrames` (McKinney, 2010; Pandas Development Team, 2020). As a result, almost all the code in `Aggregate` is domain specific.

Of course, some things `Aggregate` does not or cannot do. It does no statistical fitting: the user must supply parameters for the relevant frequency and severity distributions based on a separate analysis of available data or other industry curves. Its FFT-based algorithm can track only one variable, which makes it impossible to model a compound net of reinsurance with an aggregate limit. However, it can model the ceded position. Similarly, it cannot model a reinsurance tower where losses from one layer spill over into another but it can model disjoint layers, as typically seen in an excess of loss program. Finally, although FFT methods can model bivariate distributions, such as ceded and net, this cannot be accomplished with the current implementation.

The rest of the introduction puts `Aggregate` in context with other computational strategies in the literature and other available software packages. The remainder of the paper is structured as follows: Section 2 specifies the problem `Aggregate` solves and describes the algorithms it uses, Section 3 describes the grammar of distributions and DecL, Section 4 covers implementation, and Section 5 explains how to run DecL code and presents several illustrative examples.

1.2. Context and literature

The literature contains five principal approaches to estimating compound distributions.

1. Daykin et al. (1994), Klugman et al. (2019).
2. Panjer (1981), Bühlmann (1984), Daykin et al. (1994), Embrechts & 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), Embrechts et al. (1993), Wang (1998), Grübel and Hermesmeier (1999), Mildenhall (2005), Embrechts & Frei (2009), Shevchenko (2010).
4. Fourier transform numerical integration methods, Bohman (1969, 1974), Heckman & Meyers (1983), and 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. Grübel & Hermesmeier (1999) provided a meticulous analysis of how the FFT-based algorithm works, summarized in Section 2. Bühlmann (1984) and Embrechts and Frei (2009) compared 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 create many examples and have reproduced numerous others from published papers and have yet to find a case where it performs inadequately. Based on this experience, we recommend that FFT-based methods are adopted as standard in actuarial work.

1.3. Installation and reproducibility

`Aggregate` is class in the `aggregate` Python package. `aggregate` includes a `Severity` class, a `Portfolio` class that supports modeling with multiple compound distributions in a portfolio, including extensive pricing and capital allocation capabilities, and an `Underwriter` class that interprets DecL programs. The `build` object (see Section 5.1) is an `Underwriter` instance. The `aggregate` package was used to create all of the tables and figures in the book (Mildenhall & Major, 2022). The source code is available at https://github.com/mynl/aggregate. There is extensive documentation (Mildenhall, 2023) with the latest version hosted at https://aggregate.readthedocs.io/. The package is available from the PyPI package repository at https://pypi.org/project/aggregate/ and can be installed using `pip install aggregate==0.22.0`. Omit the version to install the latest build.

To avoid conflicts with existing installations, it is best to install into a new `Python >= 3.10` virtual environment set up as follows:

```
python -m venv path/to/your/venv
```

Next, activate your environment by:

```
path/to/your/venv/Scripts/activate
```

on Windows or:

```
source /path/to/your/venv/bin/activate
```

on Linux/Unix or MacOS. Finally, install the package:

```
pip install aggregate==0.22.0
```

All the code examples in the paper have been tested and run in such a virtual environment. Alternatively, the package can be installed and run in the cloud on Colab, https://colab.google/. Colab may report dependency resolver issues but the package does install successfully. The Colab runtime must be restarted after installation.

1.4. Similar software

Kaas et al. (2008) and Parodi (2015) presented 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., 2024) provides the same functionality in Python and includes FFT-based methods.

2. Problem and algorithms

`Aggregate` calculates the probability distribution of total subject losses from a portfolio of insurance risks using the collective risk model (Klugman et al. 2019, Chapter 9). Section 2.1 defines the insurance-related terminology necessary to specify the distribution mathematically. Section 2.2 explains how it can be approximated quickly and accurately using Fourier methods. Section 2.3 aims to demystify FFTs showing how they compute convolutions and introduce aliasing. Understanding aliasing is critical to diagnose issues that can arise with the FFT-based algorithm, and we give examples that reveal how aliasing can be exploited in certain circumstances. Section 2.4 describes different ways of discretizing distributions.

2.1. The `aggregate` calculation

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

$$Z_i = \min(y, \max(X_i - a, 0))$$

against a ground-up loss X_i. 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_i usually has a mass at zero equal to $\Pr(X_i \le a)$, complicating analysis. To remove it, let $Y_i = (Z_i \mid X_i > a)$ equal the insured loss payment, conditional on a payment being made. Severity is conditional in this way if an attachment is specified. If no attachment is specified, $Y_i = X_i$ is used directly with no conditioning. Y_i 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, Chapter 8).

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

$$A = Y_1 + \cdots + Y_N. \tag{2.1}$$

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

$$C_i = r_s \min(r_y, \max(Y_i - r_a, 0))$$

$$R_i = Y_i - C_i.$$

Here r_y is the reinsurance occurrence limit, r_a the attachment, and r_s the share of the cover purchased, $0 \leq r_s \leq 1$. 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 $r_s < 1$, the reinsurance is said to be partially placed or coinsured. `Aggregate` can model compound ceded A_C or retained A_R losses

$$A_C = C_1 + \cdots + C_N$$

$$A_R = R_1 + \cdots + R_N. \tag{2.2}$$

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. Finally, 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 multilayer programs. See Section 3.11 for more details.

2.2. Estimating compound distributions with Fourier transforms

We want a way to compute the distribution function F_A of a compound random variable given by Eq. (2.1) or (2.2). We do this by approximating 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

$$F_A(kb) := \Pr(A \leq kb) = \sum_n \Pr(A \leq kb \mid N = n) \Pr(N = n) = \sum_n F_Y^{*n}(a) \Pr(N = n),$$

where $F_Y^{*n}(a)$ denotes the distribution of $Y_1 + \cdots + Y_n$. Usually, this problem has no analytic solution because the distribution of sums of Y_i is rarely known. For example, there is no closed-form expression for the sum of two lognormals (Milevsky & Posner, 1998). However, things are more promising in the Fourier domain because the characteristic function of F_Y^{*n} is always known – it is simply the nth power of the characteristic function of F_Y.

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:

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

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 FFT 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_Y and difference to approximate the density or mass function of Y.
2. Apply the FFT to approximate ϕ_Y.
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 mass function.

The pgf is applied element-by-element in Step 3. For some Y, 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_Y directly, replacing steps 1 and 2, see Mildenhall (2023, Section 5.4.4.3) for examples.

2.3. Discrete Fourier transforms and fast Fourier transforms

This subsection defines discrete Fourier transforms (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.

2.3.1. 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 lth elements equals

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

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 lth 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 modulo 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 original shorter vectors because all the other terms are zero. `Aggregate` exploits padding to obviate the impact of aliasing.

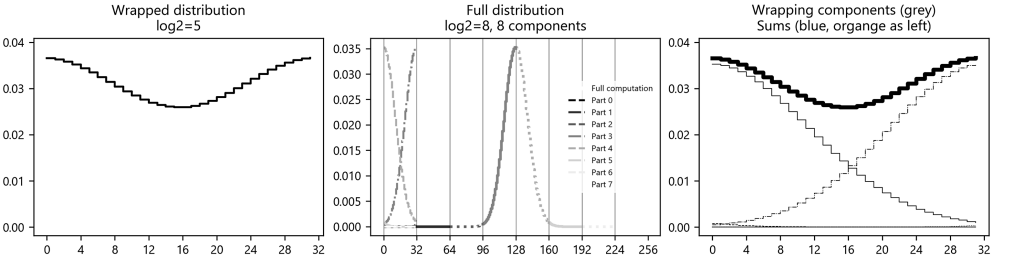


Figure 1. An apparent failure of the FFT-based method caused by aliasing (left). The middle plot shows the parts of the underlying distribution that are added by aliasing. The right plot shows these translated back to the range 0–32. This figure can be recreated by running `import aggregate.extensions` and `aggregate.extensions.fft_wrapping_illustration(ez = 1, en = 128, small2 = 5, cmap='Greys_r')`.

2.3.2. Aliasing examples

Here are two examples of aliasing. The first shows how the FFT-based algorithm can produce apparently nonsensical results. Fig. 1 shows 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 $\mathsf{x} = (0, 1, 0, \ldots, 0)$ 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 0–31 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 second example shows how we can sometimes exploit aliasing to our advantage. It is often suggested (e.g., Parodi, 2015, p. 248) 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 such 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 $10^5 \approx 2^{17}$. Fig. 2 shows the result of applying 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 still caused by aliasing but is too small to have any practical effect. It could be removed using 2^{18} buckets.

2.3.3. 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 & Tukey, 1965; Press et al., 1992). Computing the DFT Fx as a matrix multiplication should take on the order of n^2 operations. The FFT exploits a much faster approach. The kth component of Fx can be rearranged into

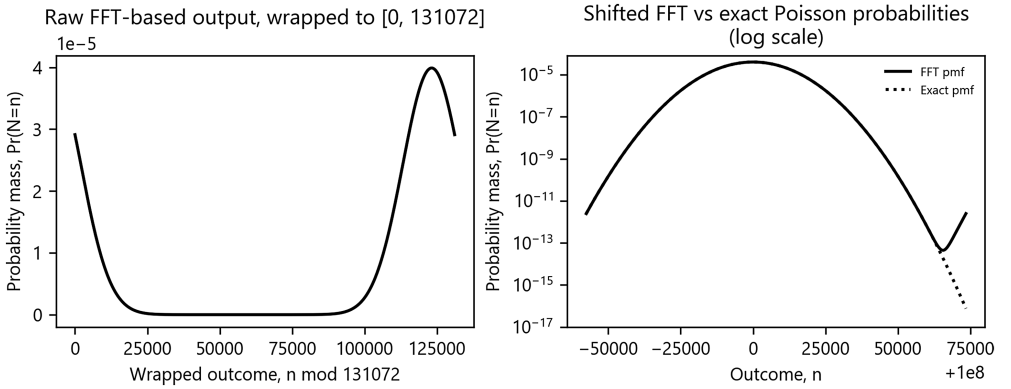


Figure 2. Exploiting aliasing to compute a Poisson distribution with a very high mean using substantially fewer buckets than is normally recommended. The actual distribution (right) can be re-assembled from the aliased output (left) by shifting. This figure can be recreated by running `aggregate.extensions.poisson_example(10**8, 17)`.

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

Define the even and odd parts $\mathsf{x}_\mathsf{e} = (x_0, x_2, \ldots, x_{n-2})$ and $\mathsf{x}_\mathsf{o} = (x_1, x_2, \ldots, x_{n-1})$. We recognize the rearranged sum as a weighted sum of the kth elements of the two smaller DFTs

$$\hat{\mathsf{x}}_k = \begin{cases} \widehat{\mathsf{x}_\mathsf{e}}_k + \omega^k\,\widehat{\mathsf{x}_\mathsf{o}}_k & k < n/2 \\ \widehat{\mathsf{x}_\mathsf{e}}_l - \omega^l\,\widehat{\mathsf{x}_\mathsf{o}}_l & k = n/2 + l \end{cases},$$

since ω^2 is an $n/2$th root of unity, and $\omega^k = -\omega^l$ implying $\omega^{2k} = \omega^{2l}$. 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.

2.4. 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 approximation 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.

2.4.1. 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 and quantile function. 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 contrast, a piecewise linear continuous approximation has a step-function pdf.

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 nonzero 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) considered a quasi-FFT-based algorithm, using discrete-continuous adjustments to reflect a piecewise linear instead of a fully discrete, cdf approximation. Reviewing 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

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

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

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

The limited expected value $\mathsf{E}[A \wedge a] = \int_0^a S_A(x)\,dx$ 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 at kb by p_k/b.

2.4.2. 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 gross loss Y and q the corresponding lower quantile function. We want to approximate F with a finite, purely discrete distribution supported at points 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 $p_0 = F(b/2)$ and probability to the $k > 1$ bucket equal to

$$p_k = \Pr((k - 1/2)b < Y \le (k + 1/2)b) = F((k + 1/2)b) - F((k - 1/2)b).$$

2. The forward difference method assigns $p_0 = F(b)$ and
$$p_k = \Pr(kb < Y \le (k+1)b) = F((k+1)b) - F(kb).$$
3. The backward difference method assigns $p_0 = F(0)$ and
$$p_k = \Pr((k-1)b < Y \le kb) = F(kb) - F((k-1)b).$$
4. The local moment matching method (Klugman et al. 2019, p. 180) ensures the discretized distribution has the same first moment as the original distribution. This method can be extended to match more moments. However, the resulting probabilities are not guaranteed to be positive per Embrechts & Frei (2009), who also report that the gain from this method is "rather modest," so we do not implement it or consider it further.

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 $p_k \ge 0$ 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 $Y_b^{(d)}$

$$\Pr(Y_b^{(d)} = kb) = p_k$$
$$F_b^{(d)}(kb) = \sum_{i \le k} p_i$$

where superscript $d = r,\ f,\ b$ 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 & Frei, 2009)

$$\begin{aligned} &F_b^{(b)} \le F \le F_b^{(f)} \quad \text{and} \quad F_b^{(b)} \le F_b^{(r)} \le F_b^{(f)}, \\ &Y_b^{(b)} \ge Y \ge Y_b^{(f)} \quad \text{and} \quad Y_b^{(b)} \ge Y_b^{(r)} \ge Y_b^{(f)}, \\ &Y_b^{(b)} \uparrow Y \quad \text{and} \quad Y_b^{(f)} \downarrow Y \text{ as } b \downarrow 0. \end{aligned} \tag{2.3}$$

$Y_b^{(b)}$, $Y_b^{(f)}$, and $Y_b^{(r)}$ converge weakly (in L^1) to Y as $b \downarrow 0$, and the same holds for a compound distribution with severity Y. These facts are needed to show the FFT algorithm works.

`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 by Embrechts & Frei (2009), Klugman et al. (2019), and Panjer (1981).

Pittarello et al. (2024) included an illustration (Fig. 1) of the different discretizations. As they explain, GEMAct offers the same discretization methods, but with one small difference. It puts mass $1 - F((m+1/2)b)$ in the last bucket to ensure the severity sums to one. In `Aggregate`, distributions can be normalized to ensure they sum to 1 as described next.

2.4.3. Truncation and normalization

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 is bounded (e.g., by a policy limit) or is unbounded but thin-tailed, there is minimal truncation error (i.e., an error caused by truncating the severity). Then, normalization is numerically cleaner and avoids issues with quantiles close to 1. However, when severity is thick-tailed, truncation error is unavoidable. It should not be normalized, as doing so can lead to unreliable results and make interpreting the estimated mean severity difficult. Without normalization, 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.

2.4.4. 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 & Hermesmeier (2000) explained 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.

2.4.5. Exponential tilting

As we have seen, the FFT-based method's circular convolution introduces aliasing error, where the probability of large losses "wrap around" and appears near the origin, polluting the probabilities of small losses, see Section 2.3.2. Grübel & Hermesmeier (1999) and Shevchenko (2010) explained how to apply exponential tilting to the severity distribution to reduce aliasing error. Exponential tilting is the same method used in exponential family distributions to adjust the mean. Tilting is also known as an exponential window (Schaller & Temnov, 2008). It is implemented using a tilt operator on discretized distributions defined by $E_\theta(\mathsf{p}) = (p_0, e^{-\theta}p_1, e^{-2\theta}p_2, \dots, e^{-(n-1)\theta}p_{n-1})$. The result is no longer a distribution (it does not sum to 1), but it can still be used as one in the FFT-based algorithm. Tilting commutes with the convolution of distributions

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

because the lth term of the convolution sum equals

$$\sum_{j+k=l} e^{-j\theta}p_j e^{-k\theta}p_k = e^{-l\theta} \sum_{j+k=l} p_j p_k.$$

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

The lth term from the circular convolution algorithm equals the true value of the lth term of the convolution plus some wrapped terms. The true term is multiplied by $e^{-l\theta}$ whereas the wrapped terms are all multiplied by at least $e^{-(l+n)\theta}$, where n is the length of `p`. Untilting multiplies them all by $e^{l\theta}$, shrinking the wrapped terms by at least $e^{-n\theta}$. The tilt parameter should be selected to that $n\theta$ is between 20 and 36 to avoid underflow. Embrechts & Frei (2009) recommended $\theta n \leq 20$. Schaller & Temnov (2008, Section 4.2) discussed other ways to select θ

Tilting is an effective way to reduce aliasing. However, once again, tilting is rarely necessary given the power of modern computers. Wang (1998) described how to pad input vectors as a more straightforward way to control aliasing, see Section 4.3. We find that padding almost always suffices to control aliasing, although `Aggregate` offers the option to apply exponential tilting if desired.

3. The grammar of distributions

Compound distributions used by practicing actuaries can be very complicated, leading to our second challenge: devising a succinct yet flexible "grammar of distributions" specification. Specification is a stand-alone problem independent of any implementation, and we solve it using a domain-specific language called DecL. DecL can express complex real-world distributions in a straightforward way, leveraging familiar insurance-domain terminology. At the same time, DecL specifies simple compounds with minimal overhead, which eases the learning curve for a new user. Any reasonably powerful compound distribution generator should be able to create any distribution that can be expressed in DecL.

`Aggregate` parses and translates DecL code into API calls, which then create approximate distributions, as explained in Section 5.1. DecL represents a mid-point between the API and a GUI-based user interface. It speaks the user's language and shields them from the intricacies of the underlying class structure and method call signatures.

Section 3.1 describes domain-specific languages and Section 3.2 the lexer. The remaining sections detail the DecL grammar. In doing so, we present the full range of compound distributions that can be approximated using `Aggregate`.

3.1. Domain-specific languages

A domain-specific language (DSL) is a programing or specification language dedicated to a particular problem domain. DSLs have a focused scope and can offer simple, expressive, and concise syntax and powerful abstractions for tasks within their domain. They often include specialized constructs and notations that are intuitive to professionals from the specific domain, even if those professionals are not primarily programmers. SQL, HTML, and CSS are examples of DSLs.

DSLs can increase productivity and accuracy in their specific domain by allowing domain experts to express concepts directly and succinctly. However, their specialized nature also means they are not suitable for general-purpose programing tasks (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. The name DecL (Dec Language) derives from an insurance policy's declarations "dec" page that spells out key coverage terms and conditions. A reinsurance placement slip performs the same functions. DecL is designed to make it easy to go from "dec page to distribution."

The DecL grammar is specified in Backus Naur form, a series of rules that show how higher-level constructs are created from lower-level ones. For example, Section 3.5 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).

The grammar 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 by Aho et al. (1986) and Levine et al. (1992).

3.2. The DecL lexer

When a DecL program is interpreted, a lexical analyzer (lexer) first breaks it into tokens and passes them to the parser, which converts them into a keyword-value argument (`**kwarg`) dictionary understood by the object API. 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 DecL lexer 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.

The lexer accepts numbers in decimal, percent, or scientific notation. A minus sign joined to a number, like −1, has a different effect than − 1 with a space, as explained in Section 3.8. 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.

3.3. The eight clause defining a compound distribution

Following the calculation in Section 2.1, the DecL grammar identifies eight clauses that define a compound distribution.

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

Clauses with an asterisk are optional. Throughout the rest of this section, terms in `<UPPER_CASE>` represent user inputs, while lowercase terms refer to language keywords. The next sections examine each clause separately.

3.4. 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 `sev`, `port` and `distortion` keywords to create severity, portfolio and spectral distortion objects. `NAME` is a string identifier, such as `Trucking` or `GL.1`. Objects can be recalled by name, see Section 3.10.

3.5. 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:

```
<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 expected loss ("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 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).
- `claim` and `claims` are interchangeable keywords and `CLAIMS` equals the expected claim count. Expected loss equals expected claim count times average severity.
- The `dfreq` discrete distribution syntax directly specifies frequency outcome and probability vectors, as described in Section 3.6. Expected loss equals the implied expected claim count times average severity.

3.6. Discrete distributions

Discrete frequencies and severities can be specified using the keywords `dfreq` and `dsev` (used in Section 3.8). 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. Nonpositive 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 distinct 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.

3.7. 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 $Y = \min(y,\ \max(X - a, 0)) \mid X > a$. If the clause is missing, `LIMIT` is treated as infinite, but `DEDUCTIBLE` is subtly different from zero, as explained in Section 3.8.

3.8. The severity clause

The severity clause specifies the unlimited ground-up loss distribution. 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. 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 severity clause can be specified in three ways. The first form enters the shape parameters directly. The second reparameterizes shape in terms of CV. It can be used for distributions with only one shape parameter or the beta distribution on $[0, 1]$, and where the first two moments exist. `aggregate` uses a formula (e.g., lognormal $\sigma = \sqrt{\log(1 + cv^2)}$, gamma $\alpha = 1/cv^2$) or, for all other distributions, the Newton–Raphson algorithm to solve for the shape parameter mapping to the requested CV and then scales to the desired mean. The third defines a discrete distribution analogously to `dfreq`. The discrete distribution is used directly.

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

- `sev` is a keyword indicating the severity specification.
- `DIST_NAME` is the `scipy.stats` continuous random variable distribution name. Common examples include `norm` Gaussian normal, `uniform`, and `expon` the exponential (with no shape parameters); `pareto`, `lognorm`, `gamma`, `invgamma`, `loggamma`, and `weibull_min` the Weibull (with one); and the `beta` and `gengamma` generalized gamma (with two). Distributions with three or more shape parameters are not supported. See Mildenhall (2023, Section 2.4.4.5) for a full list of available distributions.
- `SHAPE1`, `SHAPE2` are the required `scipy.stats` shape variables for the distribution. Shape parameters entered for zero parameter distributions are ignored.
- `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.
- `dsev` is a keyword to create discrete severity. It works in the same way as `dfreq`, see Section 3.6.

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 10 * lognorm 1.517 + 20` creates a lognormal, $10X + 20$, $X \sim \text{lognormal}(\mu = 0, \sigma = 1.517)$ with CV equal to $\sqrt{\exp(\sigma^2) - 1} = 3.0$.
- `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, whereas `sev 100 * pareto 1.3 - 100` creates a Pareto with survival function $S(x) = (100/(100+x))^{1.3}$, $x \geq 0$. Note: a negative location shift must be entered with a space since `sev pareto 1.3 -10` 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 Section 3.13. 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 nonpositive 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) standard normal as severity. All negative values are accumulated into the zero bucket, creating a mixed distribution. Consider:

```
agg NoLimitClause dfreq [1]          sev norm
agg LimitClause   dfreq [1] inf xs 0 sev norm
```

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 dfreq [1]         dsev [-1 0 1 1 2]
agg LimitClause   dfreq [1] 10 xs 0 dsev [-1 0 1 1 2]
```

`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 dfreq [1] 2 xs 1 dsev [1:3]
agg Uncond      dfreq [1] 2 xs 1 dsev [1:3] !
```

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.

3.9. The frequency clause

The frequency clause completes the specification of the frequency distribution. The clause must contain the distribution name and appropriate parameters unless `dfreq` is used in the exposure clause, in which case there is nothing else to specify, and the frequency clause is empty.

There are two types of frequency distributions: basic named distributions falling into the Panjer $(a, b, 0)$ class (Klugman et al., 2019, Section 6.5), 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 rule that frequency `SHAPE1` inputs a CV.

Zero truncated and zero modified (Klugman et al., 2019, Section 6.6) 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 claims sev gamma 2 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]`.

The second type, mixed Poisson frequency distributions, have $N \sim \text{Po}(nG)$ for a mixing distribution G with mean 1. These are appropriate for modeling larger portfolios (Mildenhall, 2017) and are specified

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

`SHAPE1` always specifies the CV of the mixing distribution. N has unconditional variance $n(1 + (cv)^2 n)$. The meaning of the second shape parameter varies. 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>`.
- `mixed sichel <SHAPE1> <SHAPE2>` is Sichel's (generalized inverse Gaussian) distribution with `<SHAPE2>` equal to the λ parameter (Johnson et al., 2005, Section 11.1.5).

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 Section 3.13 for more about mixed frequencies.

3.10. Using names

Compound and severity distributions can be recalled by name. The name is prefixed with the type. Here, the first line creates a named severity that is used in the second line.

```
sev MySev 120 * lognorm 1.8
agg Gross 10 claims 1000 xs 100 sev.MySev poisson
```

The `Gross` distribution can be recalled as `agg.Gross` and used as a subject distribution for reinsurance, see Section 3.11.

3.11. The reinsurance clauses

`Aggregate` can model nonoverlapping multilayer per-occurrence and aggregate reinsurance structures. Reinsurance is specified in a flexible 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).

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

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

The number `SHARE` specifies a share (partial placement); `so` stands for "share of". The default share is 100% if the first sub-clause is omitted. The layer transforms gross loss Y_i into

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

where $0 \le s \le 1, y, a \ge 0$ 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 inf xs 1000
```

models a 50% placement of a 250 xs 250 layer, 90% of 500 xs 500, and 100% 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 nondecreasing function of underlying losses using multiple layers in this way, capturing all reasonable indemnity functions (Huberman et al., 1983).

The concepts described so far apply to both occurrence and aggregate covers. Occurrence reinsurance is applied to individual claims and adjusts severity. It is specified before the frequency clause or after a named compound. The compound can capture the cession or losses net of the cession, Eq. (2.2). The next two examples apply reinsurance to the `Gross` distribution created in Section 3.10:

```
agg Net.1 agg.Gross occurrence ceded to 900 xs 100
agg Net.2 agg.Gross occurrence net of 75% so 750 xs 250
```

The first line models losses ceded to the 900 xs 100 layer creating a compound $A_C = \sum_{i=1}^{N} \min(900, \max(Y_i - 100, 0))$. The second models losses net of a 75% placement 750 xs 250, creating a compound $A_R = \sum_{i=1}^{N} Y_i - 0.75 \min(750, \max(Y_i - 250))$.

Aggregate reinsurance 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,

```
agg Net.3 agg.Net.2 aggregate ceded to 50% so 4000 xs 1000
```

models the cession $0.5 \min(4000, \max(A - 1000, 0))$ where A represents the subject compound distribution. When occurrence and aggregate programs are combined, the occurrence inures to the benefit of aggregate. The same syntax allows:

```
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 10,000, 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. Losses are conditional on attaching for a limit clause, 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 dfreq [1] 2 xs 1 dsev [1:3]
agg Reinsurance  dfreq [1]       dsev [1:3] occurrence ceded to 2 xs 1
```

The limit and occurrence reinsurance clauses also differ significantly in their implementation. The limit clause is implemented analytically and adjusts the cumulative distribution function and survival function 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 inconvenient to include in the validation process.

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.

3.12. 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 Section 4.6.

3.13. 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. The `Aggregate` implementation does a lot of work to implement vectorization, saving the user the need to break up mixed severities into different limit and attachment components, compute expected losses, and combine with a shared mixing variable, and this is a distinguishing feature of the implementation. A similar result could be obtained with GEMAct (Pittarello et al., 2024) but it would require manually assembling a numerical approximation to the mixed severity.

Before describing vectorization, we must present its mathematical basis. Suppose there are r families of iid gross loss 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 nontrivial, then shared mixing induces correlation between the family compounds A_i; otherwise they are independent. Standard results in the theory of compound Poisson processes (Bowers et al., 1997, Section 12.4) 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

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

Vectorized exposures and mixed severity distributions can only be used with mixed Poisson frequency distributions because they rely on this identity.

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 & 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 (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.

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

```
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. Appending a weights term to the severity clause results in each unit using the same mixed severity, creating nine components in total.

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 of vectorization enables streamlined excess of loss reinsurance exposure rating.

4. Algorithm and implementation

This section describes the `Aggregate` convolution algorithm, its implementation, what errors it introduces, and how they can be controlled.

4.1. Algorithm inputs

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

1. Ground-up severity distribution cdf and sf. See Section 3.8 for the set of allowable severity distributions and Sections 3.7 and 3.11 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 Section 3.9 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 Section 4.2.
5. Severity calculation method: round (default), forward, or backward, see Section 2.4.2.
6. Discretization calculation method: survival (default), distribution, or both, see Section 4.5.
7. Normalization: true (default) or false, see Section 2.4.3.
8. Padding parameter, an integer $d \geq 0$ with default 1, see Section 4.3.
9. Tilt parameter, a real number $\theta \geq 0$, with default 0 meaning no tilting, see Section 2.4.5.

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

4.2. Estimating the bandwidth

Correctly estimating the bandwidth is critical to obtaining accurate results from the FFT-based algorithm, as we saw in Section 2.3.2. The bandwidth needs to be small enough to capture the shape of the severity but large enough so that the range spanned by all the buckets contains the shape of the compound.

`Aggregate` employs an bandwidth 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 nonlocal behavior of thick-tailed distributions (Mandelbrot, 2013, Section 2.3.3) it is always possible to find examples where any proposed algorithm fails. Nonlocal behavior can lead to different moments being influenced by nonoverlapping 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 validation, see Section 4.7.

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. Step 9 ensures that b is a round number when $b \geq 1$ and step 10 that $b < 1$ is an exact binary float. It can cause irritating numerical issues, particularly computing quantiles, if $b < 1$ is not an exact binary float. Recall, 0.1 is not an exact binary fraction, its base 2 representation is $0.0001\overline{1001}$.

4.3. 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 & Temnov (2008) reported 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.

4.4. 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 Section 4.2.
3. Discretize severity into a vector $\mathsf{p} = (p_0, p_1, \dots, p_{n-1})$, see Section 2.4.2. 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 & Hermesmeier (1999) provided 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, Eq. (2.3). 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.

The required FFTs in steps 7 and 9 are performed by 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, as opposed to complex, numbers. The FFT routine automatically handles padding the input vector (step 6). The inverse transform returns real numbers only, so there is no need to take the real part to remove noise-level imaginary parts.

4.5. 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.

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 & Hermesmeier (1999) presented 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 Section 2.3.2.

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

1. The `distribution` 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` 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 `both` attempts to take 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 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) provided a thorough survey of FFT errors. They can result in large relative errors for small probabilities. See Wilson & Keich (2016) for examples and an approach to minimizing relative error.

4.6. 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, see Section 5.1. 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` arguments.

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` $= 0$, it is estimated using Section 4.2.
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) signaling 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.

4.7. 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, p. 153). 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. Validation 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. This test fails if severity does not have a mean.
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 the tests. Once a moment fails, no higher moments are considered. The object returns the best outcome `<Validation.NOT_UNREASONABLE: 0>` if no test fails, otherwise it describes which tests fail. For example, the compound `agg Tricky 10 claims`

`sev lognorm 3 poisson` is hard to estimate because the severity CV is over 90. After updating with default parameters, validation returns

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

indicating that it fails all moment tests.

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 Section 3.11. 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 presented automatically when the object is printed.

5. Examples and workflow

This section starts by showing how to create a simple `Aggregate` object first using the API and then using DecL, to establish the relationship between the two approaches. Section 5.3 replicate exhibits from Parodi (2015) and Grübel and Hermesmeier (1999), useful for actuaries and researchers. They confirm that `Aggregate` reproduces previously published results. Section 5.4 addresses a more complex actuarial pricing problem, using an intricate spliced mixed severity distribution from Albrecher et al. (2017). The last example, Section 5.5, is tailored to the pedagogical needs of teachers and students. Readers should 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 the calculation 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. Using the API and DecL

We show how to create the same compound object using the `Aggregate` API and then using DecL. While DecL simplifies the workflow for users, programmers may find working with the class API easier.

We want to model a book of trucking third-party liability insurance business assuming:

- Premium equals 750, and the expected loss ratio equals 67.5% (expected losses 506.25),
- 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, and
- Frequency is modeled using a Poisson distribution.

The following steps create this example using the API. The first line imports the `Aggregate` class and `qd`, a "quick display" helper function.

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

In [2]: api = Aggregate(name='Trucking',
   ...:           exp_premium=750.0, exp_lr=0.675,
   ...:           exp_attachment=0.0, exp_limit=1000.0,
   ...:           sev_name='lognorm', sev_mean=100.0, sev_cv=5.0,
   ...:           freq_name='poisson')
   ...:

In [3]: qd(api)

       E[X]    CV(X)  Skew(X)
X
Freq 6.3884 0.39564  0.39564
Sev  79.245  2.1191    3.825
Agg  506.25 0.92708   1.5644
log2 = 0, bandwidth = na, validation: n/a, not updated.
```

At this point, `api` exists as a Python object but it does not contain a numerical cdf. Note the steps taken by the constructor: it converts premium and loss ratio into an expected loss, computes the shape parameter from the CV, the limited expected value of the lognormal reflecting the policy limit and attachment, and derives expected claim count. It also computes the validation moments, as printed above by `qd(api)`. The `update` method uses the FFT-based method to make a numerical approximation to the cdf.

```
In [4]: api.update()

In [5]: qd(api)

       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.
```

After updating, the validation information, stored in the dataframe `api.describe`, compares the theoretic 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 5.8×10^{-9}, within the validation tolerance of 10^{-4} (resp. within 10^{-3}, 10^{-2} for CV and skewness; the latter errors not shown), producing a "not unreasonable" validation. Users should always review this table before using the numerical output. The display also reports the number of buckets and bandwidth.

The aggregate density, distribution, and other quantities are stored in a dataframe `api.density_df`. Using the updated object, we can request various statistics.

```
In [6]: print(f'Pr(Loss <= 500) = {api.cdf(500):.3f}, '
   ...:       f'Pr(Loss > 2000) = {api.sf(2000):.3f}\n'
   ...:       f'VaR 99% {api.q(0.99):,.0f}, VaR 99.9% {api.tvar(0.999):,.0f}')
   ...:
Pr(Loss <= 500) = 0.633, Pr(Loss > 2000) = 0.013
VaR 99% 2,082, VaR 99.9% 3,166
```

DecL insulates the nonprogramming user from the API and expresses the inputs in a more human-readable form, closer to the problem statement. The DecL code is:

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

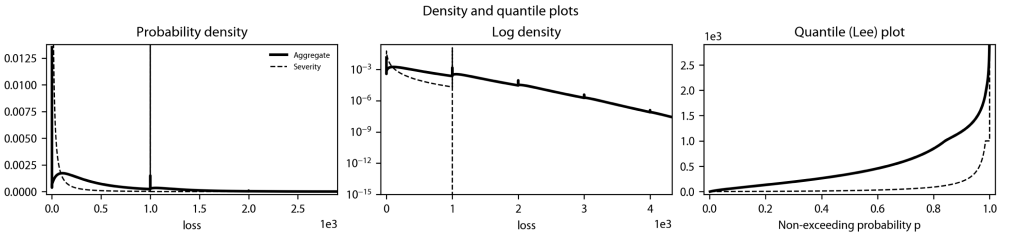
DecL code is interpreted into a Python `Aggregate` object using the `build` function. The `build` function combines the specification and update steps. It takes a DecL program, and optionally any update parameters (see Section 4.6) as keyword arguments, and returns an updated `Aggregate` object. The following Python code illustrates the recommended `Aggregate` workflow:

```
In [7]: from aggregate import build

In [8]: t = build('agg Trucking 750 premium at 0.675 lr 1000 xs 0 '
   ...:           'sev lognorm 100 cv 5 poisson')
   ...:
```

The `build` function automatically applies the `update` method, with sensible default values. The object `t` is the same as the updated `api` object and has the same validation (not shown). Note that `build` provides much more functionality: execute `x = build.show('Ca. Freq0(2|3|4)', verbose=True)` or `help(build)` to discover more.

Using the full density, it is easy to produce visualizations using standard Python functionality. For example, `t.plot()` method creates density (left) and log density (center) plots, showing severity (dashed) and compound losses (solid). 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 easiest 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. The right plot shows the severity and compound quantile functions.

```
In [9]: t.plot(); t.figure.suptitle('Density and quantile plots');
```



The API arguments created from DecL needed to recreate an object are stored in a dictionary called `spec`. The object `t2 = Aggregate(**t.spec)` replicates `t`.

Having established the relationship between the API and DecL, all subsequent examples use DecL and `build`.

5.2. Discretization method and varying bandwidth

This example reproduces an exhibit from Parodi (2015, p. 250) that explores quantiles of a compound with mean 3 Poisson frequency, and lognormal severity with $\mu = 10$ and $\sigma = 1$. Translating the Poisson-lognormal compound into DecL is straightforward: σ is the shape parameter, and $\mu = 10$ is 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 Section 2.4, and taking defaults for other update parameters produces an approximation that is not unreasonable. `Aggregate` selects a bandwidth of 70.

```
In [10]: from aggregate import build, qd

In [11]: a = build('agg Parodi 3 claims sev exp(10) * lognorm 1 poisson'
   ....:           , normalize=False)
   ....:

In [12]: 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 [13]: print(f'mean={a.agg_m:,.0f}, sd={a.agg_sd:,.0f}')
mean=108,947, sd=103,705
```

Readers should try updating the object with `normalize=True` to see the impact of this setting. The discretized severity CV and skewness statistics are closer to their actual values without normalization.

Continuing, we can update the `Aggregate` object `a` to match Parodi's calculations. 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. He discretizes severity by re-scaling the density evaluated on multiples of the bucket size. This method close to the `sev_calc='backward'`, see Section 2.4. He does not discuss padding, so we set `padding` $= 0$. Some basic Python code produces the following table.

```
In [14]: import pandas as pd

In [15]: import numpy as np

In [16]: means = ['Mean', a.est_m]; sds = ['SD', a.est_sd]

In [17]: p = np.array([.5, .75, .8, .9,  .95, .98, .99,  .995, .998, .999, .9999])

In [18]: df = pd.DataFrame({'p': p, 'Aggregate': a.q(p)})

In [19]: for bs in [100, 50, 10, 1000, '1000']:
   ....:     if type(bs) == str:
   ....:         sev_calc = 'discrete'; bs = float(bs); cn = 'h=1000r';
   ....:     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 [20]: df.loc['Mean', :] = means

In [21]: df.loc['Error on Mean', :] = ['Error on Mean'] + [f'{m / a.agg_m - 1:.2%}'
   ....:                               for m in means[1:]]
   ....:
```

```
In [22]: df.loc['SD', :] = sds

In [23]: df.loc['Error on SD', :] = ['Error on SD'] + [f'{s / a.agg_sd - 1:.2%}'␣
  for s in sds[1:]]

In [24]: df['p'] = [i if type(i) == str else (f'{i:.1%}' if i <= 0.999 else f'{i:.2
  %}') for i in df.p]

In [25]: qd(df.set_index('p'), ff=lambda x: f'{x:10,.0f}')

                Aggregate      h=100       h=50       h=10     h=1000    h=1000r
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
99.99%          1,329,510  1,329,700  1,326,800    649,890  1,332,000  1,330,000
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 Fig. 17.5 (p. 250) in Parodi's book. Interestingly, almost all the error across bandwidths > 10 is caused by the way Parodi discretizes. The last column `h = 1000r` 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.

This example highlights the importance of the discretization method, the superiority of the rounding discretization method over backwards or forwards, and the need to select the bandwidth carefully.

5.3. Aliasing, tilting, and padding

This example reproduces a table from Grübel & 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 shape 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

$$X_1 + \cdots + X_n \sim n^2 X. \tag{5.1}$$

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

The code below starts by building the `Aggregate` object with `update=False`. The first table shows the diagnostics dataframe before updating. Then `update` is applied with `bs = 1` and various other input parameters. The results across update parameters are then assembled, along with an analytic estimate of the distribution using Eq. (5.1).

```
In [26]: from scipy.stats import levy

In [27]: en = 20

In [28]: a = build(f'agg L {en} claims sev levy poisson', update=False)

In [29]: qd(a)

      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.

In [30]: bs = 1

In [31]: a.update(log2=16, bs=bs, padding=2, normalize=False, tilt_vector=None)

In [32]: df = a.density_df.loc[[1, 10, 100, 1000], ['p_total']] / a.bs

In [33]: df.columns = ['Agg pad=2']

In [34]: def exact_cdf(x, en):
   ....:     n = 5 * en
   ....:     p = np.zeros(n)
   ....:     a = np.zeros(n)
   ....:     p[0] = np.exp(-en)
   ....:     fz = levy()
   ....:     for i in range(1, n):
   ....:         p[i] = p[i-1] * en / i
   ....:         a[i] = fz.cdf((x+0.5)/i**2) - fz.cdf((x-0.5)/i**2)
   ....:     return np.sum(p * a)
   ....:

In [35]: df['True'] = [exact_cdf(i, en) for i in df.index]

In [36]: log2 = 10

In [37]: for tilt in [None, 1/1024, 5/1024, 25/1024]:
   ....:     a.update(log2=log2, bs=bs, padding=0,
   ....:              normalize=False, tilt_vector=tilt)
   ....:     if tilt is None:
   ....:         tilt = 0
   ....:     df[f'Tilt {tilt:.2g}'] = a.density_df.loc[[1, 10, 100, 1000],
   ....:                                ['p_total']]/a.bs
   ....:

In [38]: df.index = [f'{x: 6.0f}' for x in df.index]; df.index.name = 'x'

In [39]: qd(df.iloc[:, [1,0,2,3,4, 5]], ff=lambda x: f'{x:11.3e}')

            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. The 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.

This example confirms that padding is an effective way to combat aliasing. Padding is 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) presented other similar examples from Embrechts & Frei (2009), Schaller & Temnov (2008), and Shevchenko (2010).

5.4. A more complex pricing problem

This 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 parameters fit by Albrecher et al. (2017), Section 6.6, for MTPL Company A.

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 63,410, 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 [40]: from aggregate import build, qd, pprint_ex

In [41]: import pandas as pd

In [42]: pi = 0.777; alpha = np.array([0.155, 0.845]); r = np.array([1, 4])

In [43]: scale = 63410.; pareto_alpha = 1 / 0.506; t = 500000.

In [44]: wts = np.hstack((alpha * pi, 1-pi))

In [45]: shape = np.hstack((r, pareto_alpha))

In [46]: 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 [47]: 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 [48]: gross = build('agg Gross 55.27 claims sev.MTPL.A poisson'
   ....:               , bs=5000, log2=19, normalize=False)
   ....:

In [49]: 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
log2 = 19, bandwidth = 5000, validation: not unreasonable.

In [50]: df = pd.DataFrame({'p': [0.95, 0.99, 0.995]})

In [51]: df['Sev VaR'] = gross.q_sev(df.p)

In [52]: df['Alb sev'] = [1065451., 2405538., 3416123.]

In [53]: df['Agg VaR'] = gross.q(df.p)

In [54]: df['Alb agg'] = [31100000., 41420000., 48760000.]

In [55]: qd(df, ff='basic', 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 1 M xs 1 M. When reinsurance is present the validation dataframe shows theoretic gross moments and numeric net or ceded losses, making it easy to see the impact of reinsurance. This behavior is flagged by the validation result: `n/a, reinsurance`.

```
In [56]: 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 [57]: 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 [58]: df['Net Sev VaR'] = net.q_sev(df.p)

In [59]: df['Net Agg VaR'] = net.q(df.p)

In [60]: df['Occ Chg'] = df['Net Agg VaR'] / df['Agg VaR'] - 1
```

```
In [61]: qd(df.drop(columns=['Alb sev', 'Alb agg']), ff='basic', 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 [62]: 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 [63]: qd(net_agg)

           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 1.8163e+07 -0.15581      inf   0.36533              94.217
log2 = 19, bandwidth = 5000, validation: n/a, reinsurance.

In [64]: df['Net-net Agg'] = net_agg.q(df.p)

In [65]: df['Agg Chg'] = df['Net-net Agg'] / df['Agg VaR'] - 1

In [66]: qd(df.drop(columns=['Alb sev', 'Alb agg', 'Net Sev VaR',  'Occ Chg']),
   ....:     ff='basic', 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%
```

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`.

Table L and M charges (Fisher et al., 2017) compare expected losses with different occurrence and aggregate limits. Thus, by adjusting the occurrence and aggregate layers, they can be computed using this template. It can also 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 multilayer 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.

5.5. A discrete compound for educators and students

This 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 quantile function 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. The discrete 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 [67]: from aggregate import build, qd

In [68]: 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 [69]: 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.
```

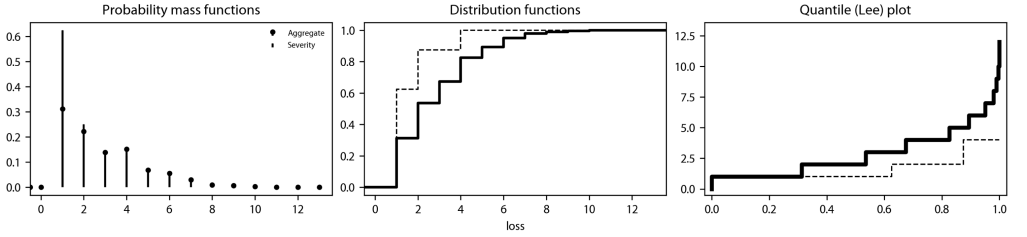
The aggregate pmf is available in the dataframe `ex.density_df`. Here are the pmf, cdf, and sf evaluated for all possible outcomes, answering question 2. The index is adjusted to an `int` to print nicely.

```
In [70]: bit = ex.density_df; bit.index = bit.index.astype(int)

In [71]: qd(bit.query('p_total > 0')[['p_total', 'F', 'S']], ff='int_ratio')

        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 [72]: 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. It introduces DecL, a domain-specific language that enables users to specify compound distributions using familiar insurance and actuarial terms. Distributions can be parameterized using the mean, CV, and other user-relevant quantities. To our knowledge, `Aggregate` is the first package to speak the user's language 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, avoiding 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.

Acknowledgements. I would like to add an acknowledgement as follows: I would like to thank all users of this software and the reviewers for their helpful comments.

Data availability statement. The data and code that support the findings of this study are openly available on GitHub at https://github.com/mynl/aggregate. The results contained in the manuscript are reproducible in a virtual environment, excluding environment-specific numerical errors. These discrepancies do not affect the overall validity of the results. The repository is registered with the unique Zenodo DOI reference number https://doi.org/10.5281/zenodo.10557198.

Funding statement. This work received no specific grant from any funding agency, or commercial or not-for-profit organization.

Competing interests. The author declares no competing interests.

References

Aho, **A. V.**, **Sethi**, **R.**, & **Ullman**, **J. D.** (1986). *Compilers, principles, techniques*. Addison Wesley.

Albrecher, **H.**, **Beirlant**, **J.**, & **Teugels**, **J. L.** (2017). *Reinsurance: Actuarial and statistical aspects*. John Wiley & Sons.

Bahnemann, **D.** (2015). Distributions for Actuaries. Casualty Actuarial Society Mongraphs No. 2. ISBN 9780962476280. www.casact.org.

Bear, **R. A.**, & **Nemlick**, **K. J.** (1990). Pricing the impact of adjustable features and loss sharing provisions of reinsurance treaties. *Proceedings of the Casualty Actuarial Society*, **77**(147), 86–87. doi: 10.1016/0167-6687(93)91078-9.

Beazely, **D.** (2022). Sly (sly lex-yacc). https://github.com/dabeaz/sly.

Bertram, **J.** (1981). Numerische berechnung von gesamtschadenverteilungen. *Blätter der DGVFM*, **15**(2), 175–194.

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

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

Bowers, **N.**, **Gerber**, **H.**, **Hickman**, **J.**, **Jones**, **D.**, & **Nesbitt**, **C.** (1997). *Actuarial mathematics*. Society of Actuaries. doi: 10.2307/253313

Brisebarre, **N.**, **Joldeş**, **M.**, **Muller**, **J. M.**, **Naneş**, **A. M.**, & **Picot**, **J.** (2020). Error analysis of some operations involved in the Cooley-Tukey fast fourier transform. *ACM Transactions on Mathematical Software*, **46**(2), 1–27. doi: 10.1145/3368619.

Bühlmann, **H.** (1984). Numerical evaluation of the compound Poisson distribution: Recursion or fast fourier transform? *Scandinavian Actuarial Journal*, **1984**(2), 116–126. doi: 10.1080/03461238.1984.

Clark, **D. R.** (2014). Basics of Reinsurance Pricing Actuarial Study Note. CAS Study Note. http://www.casact.org/library/studynotes/Clark

Cooley, **J. W.**, & **Tukey**, **J. W.** (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, **19**(90), 297–301. doi: 10.2307/2003354.

Corro, **D.**, & **Tseng**, **Y.-c.** (2021). NCCI's 2014 Excess loss factors. *Variance*, **14**(1), 1–28.

Daykin, **C. D.**, **Pentikainen**, **T.**, & **Pesonen**, **M.** (1994). *Practical risk theory for actuaries*. Chapman and Hall.

Dutang, **C.**, **Goulet**, **V.**, & **Pigeon**, **M.** (2008). Actuar: An R package for actuarial science. *Journal of Statistical Software*, **25**(7), 1–37. doi: 10.18637/jss.v025.i07.

Embrechts, **P.**, & **Frei**, **M.** (2009). Panjer recursion versus FFT for compound distributions. *Mathematical Methods of Operations Research*, **69**(3), 497–508. doi: 10.1007/s00186-008-0249-2.

Embrechts, **P.**, **Grübel**, **R.**, & **Pitts**, **S. M.** (1993). Some applications of the fast fourier transform algorithm in insurance mathematics. *Statistica Neerlandica*, **47**(1), 59–75. doi: 10.1111/j.1467-9574.1993.tb01406.x

Embrechts, **P.**, **Klüppelberg**, **C.**, & **Mikosch**, **T.** (2013). *Modelling extremal events: For insurance and finance*, vol. **33**. Springer Science & Business Media.

Fisher, **G. K.**, **McTaggart**, **L.**, **Petker**, **J.**, & **Pettingell**, **R.** (2017). Individual Risk Rating. Technical Report April, Casualty Actuarial Science.

Frees, **E.** (2018). Loss Data Analytics. https://openacttexts.github.io/Loss-Data-Analytics/. arXiv:1808.06718.

Grübel, **R.**, & **Hermesmeier**, **R.** (1999). Computation of compound distributions I: Aliasing errors and exponential tilting. *Astin Bulletin*, **29**(2), 197–214. doi: 10.2143/AST.29.2.504611.

Grübel, **R.**, & **Hermesmeier**, **R.** (2000). Computation of compound distributions II: Discretization errors and Richardson extrapolation. *ASTIN Bulletin*, **30**(2), 309–332. doi: 10.2143/AST.30.2.504638.

Heckman, **P. E.**, & **Meyers**, **G. G.** (1983) The calculation of aggregate loss distributions from claim severity and claim count distributions. In: *Proceedings of the casualty actuarial society* (pp. 49–66).

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

Huberman, **G.**, **Mayers**, **D.**, & **Smith**, **C. W.** (1983). Optimal insurance policy indemnity schedules. *The Bell Journal of Economics*, **14**(2), 415. doi: 10.2307/3003643.

Hürlimann, **W.** (1986). Error bounds for stop-loss premiums calculated with the fast fourier transform. *Scandinavian Actuarial Journal*, **1986**(2), 107–113. doi: 10.1080/03461238.1986.

Johnson, **N. L.**, **Kotz**, **S.**, & **Kemp**, **A. W.** (2005). *Univariate discrete distributions* (3rd ed.). John Wiley & Sons.

Kaas, **R.**, **Goovaerts**, **M.**, **Dhaene**, **J.**, & **Denuit**, **M.** (2008). *Modern actuarial risk theory*. Springer. ISBN 978-3-540-70992-3. arXiv:1011.1669v3. doi: 10.1007/978-3-540-70998-5.

Klugman, **S. A.**, **Panjer**, **H. H.**, & **Willmot**, **G. E.** (2019). *Loss models: From data to decisions*, vol. **715** (5th ed.). John Wiley & Sons.

Levine, **J. R.**, **Mason**, **T.**, & **Brown**, **D.** (1992). *lex & yacc*. O'Reilly.

Mandelbrot, **B. B.** (2013) *Fractals and scaling in finance: Discontinuity, concentration, risk*, Selecta volume E. Springer Science & Business Media.

Marc Goovaerts, **F. E. De V.**, & **Haezendonck**, **J.** (1984). *Insurance premiums*. North-Holland.

McKinney, **W.** (2010). Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference*, **1**(Scipy), 56–61. doi: 10.25080/majora-92bf1922-00a.

Mernik, **M.**, **Heering**, **J.**, & **Sloane**, **A. M.** (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, **37**(4), 316–344. doi: 10.1115/IPC2010-31470.

Mildenhall, **S. J.** (2005). Correlation and aggregate loss distributions with an emphasis on the iman-conover method. *Casualty Actuarial Society Forum*. https://www.casact.org/sites/default/files/database/forum

Mildenhall, **S. J.** (2017). Actuarial geometry. *Risks*, **5**(2), 31. doi: 10.3390/risks5020031.

Mildenhall, **S. J.** (2023). Aggregate documentation. URL: https://aggregate.readthedocs.io/&/downloads/en/latest/pdf/.

Mildenhall, **S. J.**, & **Major**, **J. A.** (2022). *Pricing insurance risk: Theory and practice*. John Wiley & Sons, Inc.

Milevsky, **M. A.**, & **Posner**, **S. E.** (1998). Asian options, the sum of Lognormals, and the reciprocal Gamma distribution. *Journal of Financial and Quantitative Analysis*, **33**(3), 409–422.

Pandas-dev/pandas:pandas (2020). Pandas-dev/pandas: pandas. doi: 10.5281/zenodo.3509134.

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

Parodi, **P.** (2015). *Pricing in general insurance*. CRC Press. file:///C:/Users/youhe/Downloads/kdoc_o_00042_01.pdf. ISBN 9781466581487.

Pittarello, **G.**, **Luini**, **E.**, & **Marchione**, **M. M.** (2024). GEMAct: a Python package for non-life (re)insurance modelling. *Annals of Actuarial Science*, 4–11. doi: 10.1017/S1748499524000022.

Press, **W. H.**, **Teukolsky**, **S. A.**, **Vetterling**, **W. T.**, & **Flannery**, **B. P.** (1992). *Numerical recipes in C* (2nd ed.). Cambridge University Press. ISBN 0521431085.

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

Rockafellar, **R. T.**, & **Uryasev**, **S.** (2002). Conditional value-at-risk for general loss distributions. *Journal of Banking & Finance*, **26**(7), 1443–1471.

Schaller, **P.**, & **Temnov**, **G.** (2008). Efficient and precise computation of convolutions: Applying FFTs to heavy tailed distributions. *Computational Methods in Applied Mathematics*, **8**(2), 187–200.

Shevchenko, **P. V.** (2010). Calculation of aggregate loss distributions. *Journal of Operational Risk*, **5**(2), 3–40. arXiv:1008.1108.

Strang, **G.** (1986). *Introduction to applied mathematics*. Wellesley-Cambridge Press.

Van Rossum, **G.**, & **Drake**, **F. L.** (2009). *Python 3 Reference manual*). CreateSpace. ISBN 1441412697.

Virtanen, **P.**, **Gommers**, **R.**, **Oliphant**, **T. E.**, **Haberland**, **M.**, **Reddy**, **T.**, **Cournapeau**, **D.**, **Burovski**, **E.**, **Peterson**, **P.**, **Weckesser**, **W.**, **Bright**, **J.**, **van der Walt**, **S. J.**, **Brett**, **M.**, **Wilson**, **J.**, **Jarrod Millman**, **K.Mayorov**, **N.**, **Nelson**, **A. R. J.**, **Jones**, **E.**, **Kern**, **R.**, **Eric Larson**, **C. J. C.**, **İ.**, **Polat**,. . .,**Pedregosa**, **F.** (2020). Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, **17**(3), 261–272. doi: 10.1038/s41592-019-0686-2.

Wang, **S. S.** (1998) Aggregation of correlated risk portfolios: Models and algorithms. In *Proceedings of the casualty actuarial society* (pp. 848–939). http://www.casact.com/pubs/proceed/proceed98/980848.pdf

Wilson, **H.**, & **Keich**, **U.** (2016). Accurate pairwise convolutions of non-negative vectors via FFT. *Computational Statistics and Data Analysis*, **101**, 300–315. doi: 10.1016/j.csda.2016.03.010.

Zhu, **L.** (2011) Introduction to increased limit factors. In *CAS ratemaking seminar* (pp. 1–31).

Cite this article: Mildenhall S (2024). Aggregate: fast, accurate, and flexible approximation of compound probability distributions, *Annals of Actuarial Science*, 1–40. https://doi.org/10.1017/S1748499524000216