Annals of Actuarial Science (2024), **18**, pp. 342–378
doi:10.1017/S1748499524000022

ACTUARIAL SOFTWARE

GEMAct: a Python package for non-life (re)insurance modeling

Gabriele Pittarello[1], Edoardo Luini[2] and Manfred Marvin Marchione[1]

[1]Università 'La Sapienza', Rome, Italy; and [2]Università Cattolica del Sacro Cuore, Milan, Italy
Corresponding author: Gabriele Pittarello; Email: gabriele.pittarello@uniroma1.it

(Received 07 April 2023; revised 05 January 2024; accepted 10 January 2024; first published online 14 February 2024)

Abstract

This paper introduces gemact, a Python package for actuarial modeling based on the collective risk model. The library supports applications to risk costing and risk transfer, loss aggregation, and loss reserving. We add new probability distributions to those available in scipy, including the (a, b, 0) and (a, b, 1) discrete distributions, copulas of the Archimedean family, the Gaussian, the Student t and the Fundamental copulas. We provide an implementation of the AEP algorithm for calculating the cumulative distribution function of the sum of dependent, nonnegative random variables, given their dependency structure specified with a copula. The theoretical framework is introduced at the beginning of each section to give the reader with a sufficient understanding of the underlying actuarial models.

Keywords: Insurance; collective risk model; risk costing; loss aggregation; claims reserving; Python

1. Introduction

In non-life insurance, the accurate representation and quantification of future losses is a foundational task, central to several areas ranging from pricing and reserving to risk management. Indeed, the actuarial literature is rich in models that are relevant in such applications. Among those, the collective risk model has been widely studied as it is mathematically tractable, it requires little and general information, and it can be efficiently implemented (Embrechts & Frei, 2009; Klugman *et al.*, 2012; Parodi, 2014). In particular, by knowing the frequency and severity distributions of the losses, it is possible to compute the distribution of the aggregate (total) losses. The collective risk model represents the common thread of this work, and we developed gemact to provide a collection of tools for (re)insurance modeling under a unified formal framework.

After a brief discussion on how to install the software in Section 2, we introduce the statistical framework of the collective risk model in Section 3. There, we define an aggregate loss distribution as a random sum of i.i.d. random variables, which can be computed using the recursive formula (Panjer, 1981), the discrete Fourier transform (DFT) (Bühlmann, 1984; Grübel & Hermesmeier, 1999; Wang, 1998), and a Monte Carlo (MC) simulation approach (Klugman *et al.*, 2012, p. 467). Once the aggregate loss distribution is available, its expected value can be used for costing purposes. In this respect, the package supports typical coverage modifiers like (individual and aggregate) deductibles, limits, and reinstatements (see Sundt, 1990). Also, we consider different methods for the discretization of continuous distributions (Gerber, 1982).

Often, it is necessary to model the sum of a fixed number of dependent random variables. In order to do so, in Section 4, we introduce the AEP algorithm (Arbenz *et al.*, 2011) and a MC simulation approach for evaluating the cumulative distribution function of a sum of random variables

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

with a given dependency structure. The dependency structure can be specified with the copulas we implemented. These are listed in Section A and include copulas of the Archimedean family, the Gaussian, the Student t, and the Fundamental copulas (Nelsen, 2007).

Lastly, assuming a collective risk model holds for the cells of a loss development triangle, it is possible to define the stochastic claims reserving model in Ricotta & Clemente (2016), Clemente *et al.* (2019). In this case, the user obtains information on the frequency and severity parameters of the cells from the Fisher–Lange method (Fisher & Lange, 1973). Both these approaches are described in Section 5.

1.1 Context, scope, and contributions

In the recent years, programming languages and statistical computing environments, such as Python (Van Rossum & Drake, 2009) and R (R Core Team, 2017), have become increasingly popular (Ozgur *et al.*, 2022). Currently, coding skills form part of the body of knowledge of actuaries and actuarial science researchers. In R, an extensive implementation for aggregate loss modeling based on the collective risk theory is the actuar package (Dutang *et al.*, 2008, 2022). An available library in Python is aggregate, which implements the computation of compound probability distributions via fast Fourier transform (FFT) convolution algorithm (Mildenhall, 2022). This package employs a specific grammar for the user to define insurance policy declarations and distribution calculation features. Direct access to its objects and their components is also possible.

With regard to claims reserving, chainladder offers in Python standard aggregate reserving techniques, like deterministic and stochastic chain-ladder methods, the Bornhuetter–Ferguson model, and the Cape Cod model (Bogaardt, 2022). This package is available in R and Python (Gesmann *et al.*, 2022). Furthermore, apc provides the family of age-period-cohort approaches for reserving. This package is also available in both the above-mentioned programming languages (Nielsen, 2015).

When it comes to the topic of dependence modeling via copulas, in Python one can use the copulas and copulae packages (Bok, 2022; Lab, 2022). Similarly, copula features in R are implemented in copula; see the package and its extensions in Jun Yan (2007), Kojadinovic & Yan (2010), and Hofert & Mächler (2011).

In this manuscript, we present an open-source Python package that extends the existing software tools available to the actuarial community. Our work is primarily aimed at the academic audience, who can benefit from our implementation for research and teaching purposes. Nonetheless, our package can also support non-life actuarial professionals in prototypes modeling, benchmarking and comparative analyses, and ad hoc business studies.

From the perspective of the package design, gemact adopts an explicit, direct, and integrated object-oriented programming (OOP) paradigm. In summary, our goal is to provide:

- A computational software environment that gives users control over mathematical aspects and actuarial features, enabling the creation of models tailored to specific needs and requirements.
- An object-oriented system whose elements (i.e., objects, methods, and attributes) can be accessed and managed via our API in such a way as to be interactive and suitable for users familiar with OOP designs and with the underlying modeling framework.
- A collection of extensible libraries to make gemact survive over time. Our package is designed in an attempt to be easily extended or integrated with new functionalities and modules and to respect the attributes that qualify extensible software (Johansson & Löfgren, 2009). Namely, a modification to the functionalities should involve the least number of changes to the least number of possible elements (modifiability, Bass *et al.*, 2003, p. 137), the addition of new requirements should not raise new errors (maintainability, Sommerville, 2011, p. 25), and the

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

system should be able to expand in a chosen dimension without major modifications to its architecture (scalability, Bondi, 2000).

From the perspective of actuarial advancements and developments, gemact provides an implementation to established algorithms and methodologies. In particular, our package:

- Implements (a, b, 0) and (a, b, 1) classes of distributions for describing the loss frequency (Klugman *et al.*, 2012, p. 505), and further continuous distributions to model the loss severity, like the generalized beta (Klugman *et al.*, 2012, p. 493). Additional details can be found in Section A. Moreover, it integrates these into scipy distributions (Virtanen *et al.*, 2020).
- Offers the first open-source software implementation of the AEP algorithm (Arbenz *et al.*, 2011) for evaluating the cumulative distribution function of a sum of random variables with a given dependency structure specified via a copula.
- Includes the Student t copula and a method for numerically approximating its cumulative distribution function (Genz & Bretz,1999, 2002).
- Implements the stochastic claims reserving model described by Ricotta & Clemente (2016) and Clemente *et al.* (2019) based on the collective risk model apparatus.

2. Installation

The production version of the package is available on the Python Package Index (PyPi). Users can install gemact via pip, using the following command in the operating system command line interface.

```
pip install gemact==1.2.1
```

Examples on how to get started and utilize objects and functionalities of our package will be shown below. This work refers to production version `1.2.1`.

Furthermore, the developer version of gemact can be found on GitHub at:
https://github.com/gpitt71/gemact-code
Additional resources on our project, including installation guidelines, API reference, technical documentations, and illustrative examples, can be found at:
https://gem-analytics.github.io/gemact/

3. Loss model

Within the framework of the collective risk model (Embrechts & Frei, 2009), all random variables are defined on some fixed probability space $(\Omega, \mathcal{F}, P)$. Let

- N be a random variable taking values in $\mathbb{N}_0$ representing the claim frequency.
- $\{Z_i\}_{i\in\mathbb{N}}$ be a sequence of i.i.d nonnegative random variables independent of N; Z is the random variable representing the individual (claim) loss.

The aggregate loss X, also referred to as aggregate claim cost, is

$$X = \sum_{i=1}^{N} Z_i, \tag{1}$$

with $\sum_{i=1}^{0} Z_i = 0$. Details on the distribution functions of N, Z, and X are discussed in the next sections. Equation (1) is often referred to as the frequency–severity loss model representation. This can encompass common coverage modifiers present in (re)insurance contracts (Parodi, 2014, p. 50). More specifically, let us consider:

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

- For $a \in [0, 1]$, the function Q_a apportioning the aggregate loss amount:

$$Q_a(X) = aX. \tag{2}$$

- For $c, d \geq 0$, the function $L_{c,d}$ applied to the individual claim loss:

$$L_{c,d}(Z_i) = \min \left\{ \max \left\{ 0, Z_i - d \right\}, c \right\}. \tag{3}$$

  Herein, for each and every loss, the excess to a *deductible* d (sometimes referred to as *priority*) is considered up to a *cover* or *limit* c. In line with Albrecher *et al.* (2017, p. 34), we denote $[d, d + c]$ as *layer*. An analogous notation is found in Ladoucette & Teugels (2006) and Parodi (2014, p. 46). Similarly to the individual loss Z_i, Equation (3) can be applied to the aggregate loss X.

Computing the aggregate loss distribution is relevant for several actuarial applications (Parodi, 2014, p. 93). The gemact package provides various methods for calculating the distribution of the loss model in Equation (1) that allow the transformations of Equations (2) and (3) and their combinations to be included.

3.1 Risk costing

In this section, we describe an application of the collective risk model of Equation (1). The expected value of the aggregate loss of a portfolio constitutes the building block of an insurance tariff. This expected amount is called pure premium or loss cost, and its calculation is referred as *risk costing* (Parodi, 2014, p. 282). Insurers frequently cede parts of their losses to reinsurers, and risk costing takes this transfers into account. Listed below are some examples of basic reinsurance contracts whose pure premium can be computed using gemact.

- The *Quota Share (QS)*, where a share a of the aggregate loss ceded to the reinsurance (along with the respective premium) and the remaining part is retained:

$$\mathrm{P}^{QS} = \mathbb{E}\left[Q_a\left(X\right)\right]. \tag{4}$$

- The *Excess-of-loss (XL)*, where the insurer cedes to the reinsurer each and every loss exceeding a deductible d, up to an agreed limit or cover c, with $c, d \geq 0$:

$$\mathrm{P}^{XL} = \mathbb{E}\left[\sum_{i=1}^{N} L_{c,d}(Z_i)\right]. \tag{5}$$

- The *Stop Loss (SL)*, where the reinsurer covers the aggregate loss exceedance of a (aggregate) deductible v, up to a (aggregate) limit or cover u, with $u, v \geq 0$:

$$\mathrm{P}^{SL} = \mathbb{E}\left[L_{u,v}(X)\right]. \tag{6}$$

The model introduced by Equation (1) and implemented in gemact can be used for costing contracts like the *XL with Reinstatements (RS)* in Sundt (1990). Assuming the aggregate cover u is equal to $(K + 1)c$, with $K \in \mathbb{Z}^{+}$:

$$\mathrm{P}^{RS} = \frac{\mathbb{E}\left[L_{u,v}(X)\right]}{1 + \frac{1}{c}\sum_{k=1}^{K} l_k \mathbb{E}\left[L_{c,(k-1)c+v}(X)\right]}, \tag{7}$$

where K is the number of reinstatement layers and $l_k \in [0, 1]$ is the reinstatement premium percentage, with $k = 1, \ldots, K$. When $l_k = 0$, the k-th resinstatement is said to be free. In detail, the logic we implemented implies that whenever a layer is used the cedent pays a reinstatement premium, that is, $l_k \mathrm{P}^{RS}$, and the cover c is thus reinstated. The reinstatement premium will usually be paid in proportion to the amount that needs to be reinstated (Parodi, 2014, p. 52). In practice, the reinstatement premium percentage l_k is a contractual element, given a priori as a percentage of the

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

premium paid for the initial layer. In fact, in gemact l_k is provided by the user. The mathematics behind the derivation of P^{RS} is beyond the scope of this manuscript; the interested reader can refer to Sundt (1990), Parodi (2014, p. 325), and Antal (2009, p. 57).

3.2 Computational methods for the aggregate loss distribution

The cumulative distribution function (cdf) of the aggregate loss in Equation (1) is

$$F_X(x) = P[X \leq x] = \sum_{k=0}^{\infty} p_k F_Z^{*k}(x) \tag{8}$$

where $p_k = P[N = k]$, $F_Z(x) = P\,[Z \leq x]$ and $F_Z^{*k}(x) = P\,[Z_1 + \ldots + Z_k \leq x]$.

Moreover, the characteristic function of the aggregate loss $\phi_X : \mathbb{R} \to \mathbb{C}$ can be expressed in the form:

$$\phi_X(t) = \mathcal{P}_N\left(\phi_Z(t)\right), \tag{9}$$

where $\mathcal{P}_N(t) = \mathrm{E}\left[t^N\right]$ is the probability generating function of the frequency N and $\phi_Z(t)$ is the characteristic function of the severity Z (Klugman *et al.*, 2012, p. 153).

The distribution in Equation (8), except in a few cases, cannot be computed analytically, and its direct calculation is numerically expensive (Parodi, 2014, p. 239). For this reason, different approaches have been analyzed to approximate the distribution function of the aggregate loss, including parametric and numerical quasi-exact methods (for a detailed treatment refer to Shevchenko, 2010). Among the latter, gemact implements *MC simulation* (Klugman *et al.*, 2012, p. 467), *DFT* (Bühlmann, 1984; Grübel & Hermesmeier, 1999; Wang, 1998), and the so-called *recursive formula* (Panjer, 1981). A brief comparison of accuracy, flexibility, and speed of these methods can be found in Parodi (2014, p. 260) and Wüthrich (2023, p. 127). This section details these last two computational methods based on discrete mathematics to approximate the aggregate loss distribution.

Henceforth, let us consider, for $j = 0, 1, 2, \ldots, m - 1$ and $h > 0$, an arithmetic severity distribution with probability sequence:

$$\{\mathbf{f}\} = \{f_0, f_1, \ldots, f_{m-1}\},$$

where $f_j = P[Z = j \cdot h]$. The discrete version of Equation (8) becomes

$$g_s = \sum_{k=0}^{\infty} p_k f_s^{*k},$$

where $g_s = P[X = s]$ and

$$f_s^{*j} := \begin{cases} 1 & \text{if } k = 0 \text{ and } s = 0 \\ 0 & \text{if } k = 0 \text{ and } s \in \mathbb{N} \\ \sum_{i=0}^{s} f_{s-i}^{*(k-1)} f_i & \text{if } k > 0. \end{cases}$$

3.2.1 Discrete fourier transform

The DFT of the severity $\{\mathbf{f}\}$ is, for $k = 0, \ldots, m - 1$, the sequence

$$\{\hat{\mathbf{f}}\} = \{\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{m-1}\},$$

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

where

$$\widehat{f}_k = \sum_{j=0}^{m-1} f_j e^{\frac{2\pi ikj}{m}}. \tag{10}$$

The original sequence can be reconstructed with the inverse DFT:

$$f_j = \frac{1}{m} \sum_{k=0}^{m-1} \widehat{f}_k e^{-\frac{2\pi ikj}{m}}.$$

The sequence of probabilities $\{\mathbf{g}\} = \{g_0, g_1, \ldots, g_{m-1}\}$ can be approximated taking the inverse DFT of

$$\{\hat{\mathbf{g}}\} := \mathcal{P}_N\left(\{\hat{\mathbf{f}}\}\right). \tag{11}$$

The original sequence can be computed efficiently with a FFT algorithm, when m is a power of 2 (Embrechts & Frei, 2009).

3.2.2 Recursive formula

Assume that the frequency distribution belongs to the $(a, b, 0)$ class, that is, for $k \geq 1$ and $a, b \in \mathbb{R}$:

$$p_k = \left(a + \frac{b}{k}\right) p_{k-1}. \tag{12}$$

Here, p_0 is an additional parameter of the distribution (Klugman *et al.*, 2012, p. 505). The $(a, b, 0)$ class can be generalized to the $(a, b, 1)$ class assuming that the recursion in Equation (12) holds for $k = 2, 3, \ldots$

The recursive formula was developed to compute the distribution of the aggregate loss when the frequency distribution belongs to the $(a, b, 0)$ or the $(a, b, 1)$ class. The sequence of probabilities $\{\mathbf{g}\}$ can obtained recursively using the following formula:

$$g_s = \frac{\left[p_1 - (a+b)p_0\right] f_s + \sum_{j=1}^{s} (a + bj/s) f_j g_{s-j}}{1 - af_0}, \tag{13}$$

with $1 \leq s \leq m-1$ and given the initial condition $g_0 = \mathcal{P}_N\left(f_0\right)$.

3.3 Severity discretization

The calculation of the aggregate loss with DFT or with the recursive formula requires an arithmetic severity distribution (Embrechts & Frei, 2009). Conversely, the severity distribution in Equation (1) is often calibrated on a continuous support. In general, one needs to choose a discretization approach to arithmetize the original distribution. This section illustrates the methods for the discretization of a continuous severity distribution available in the gemact package.

Let $F_z : \mathbb{R}^+ \to [0, 1]$ be the cdf of the distribution to be discretized. For a bandwidth, or discretization step, $h > 0$ and an integer m, a probability f_j is assigned to each point hj, with $j = 0, \ldots, m-1$. Four alternative methods for determining the values for f_j are implemented in gemact.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

1. The method of *mass dispersal*:

$$f_j = \begin{cases} F_Z\left(\frac{h}{2}\right) & j = 0 \\ F_Z\left(hj + \frac{h}{2}\right) - F_Z\left(hj - \frac{h}{2}\right) & j = 1, \ldots, m-2 \\ 1 - F_Z\left(hj - \frac{h}{2}\right) & j = m-1. \end{cases}$$

2. The method of the *upper discretization*:

$$f_j = \begin{cases} F_Z\left(hj + h\right) - F_Z\left(hj\right) & j = 0, \ldots, m-2 \\ 1 - F_Z\left(hj\right) & j = m-1. \end{cases}$$

3. The method of the *lower discretization*:

$$f_j = \begin{cases} F_Z\left(0\right) & j = 0 \\ F_Z\left(hj\right) - F_Z\left(hj - h\right) & j = 1, \ldots, m-1. \end{cases}$$

4. The method of *local moment matching*:

$$f_j = \begin{cases} 1 - \frac{\mathbb{E}[Z \wedge h]}{h} & j = 0 \\ \frac{2\mathbb{E}[Z \wedge hj] - \mathbb{E}[Z \wedge h(j-1)] - \mathbb{E}[Z \wedge h(j+1)]}{h} & j = 1, \ldots, m-1 \end{cases}$$

where $\mathbb{E}[Z \wedge h] = \int_{-\infty}^{h} t\, dF_z(t) + h[1 - F_z(h)]$.

Fig. 1 illustrates and graphically contrasts the four different discretization techniques.

The default discretization method in gemact is the method of the mass dispersal, in which each hj point is assigned with the probability mass of the h-span interval containing it (Fig. 1 top left). The upper discretization and lower discretization methods generate, respectively, pointwise upper and lower bounds to the true cdf. Hence, these can be used to provide a range where the original F_z is contained (see Fig. 1, top right and bottom left graphs). The method of local moment matching (Fig. 1 bottom right) allows the moments of the original distribution F_z to be preserved in the arithmetic distribution. A more general definition of this approach can be found in Gerber (1982). We limited this approach to the first moment as, for higher moments, it is not well defined and it can potentially lead to a negative probability mass on certain lattice points (Embrechts & Frei, 2009). The method of local moment matching shall be preferred for large bandwidths. However, considering that there is no established analytical procedure to determine the optimal step, we remark that the choice of the discretization step is an educated guess and it really depends on the problem at hand. In general, h should be chosen such that it is neither too small nor too large relative to the severity losses. In the first case, the hj points are not sufficient to capture the full range of the loss amount and the probability in the tail of the distribution exceeding the last discretization node $h(m-1)$ is too large. In the second case, the granularity of the severity distribution is not sufficient, and small losses are over-approximated. Additional rules of thumb and guidelines for the choice of discretization parameters can be found in Parodi (2014, p. 248). For example, one option is to perform the calculation with decreasing values of h and check, graphically or according to a predefined criterion, whether the aggregate distribution changes substantially (Embrechts & Frei, 2009). The reader should refer to Klugman *et al.* (2012, p. 179) and Embrechts & Frei (2009) for a more detailed treatment and discussion of discretization methods.

The above-mentioned discretization methods are modified accordingly to reflect the cases where the transformation of Equation (3) is applied to the severity (Klugman *et al.*, 2012, p. 517). Below, we illustrate how to perform severity discretization in gemact.

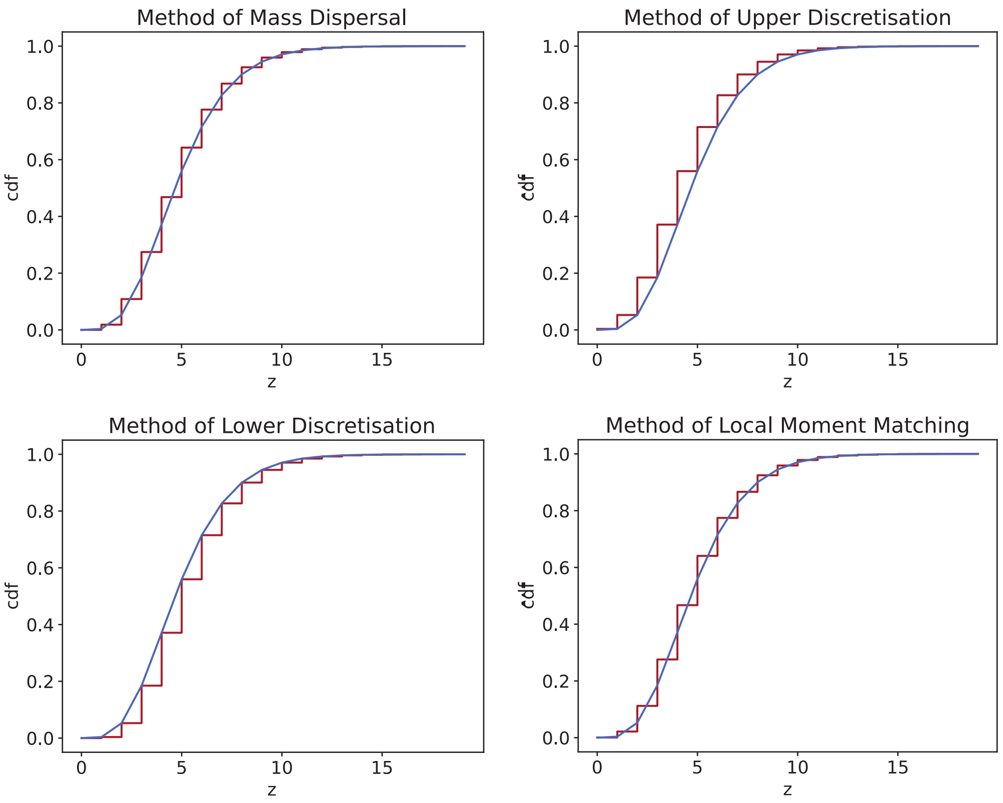


Figure 1. Illustration of the discretization methods applied to a gamma(a = 5) severity. The graphs compare the original cdf (blue line) and the discretized (red line) cdf for mass dispersal (top left), upper discretization (top right), lower discretization (bottom left), and local moment matching (bottom right) methods. No coverage modifiers are present.

Once a continuous distribution is selected from those supported in our package (see Section A), the severity distribution is defined via the `Severity` class.

```
>>> from gemact.lossmodel import Severity
>>> severity = Severity(dist='gamma', par={'a': 5})
```

The `dist` argument contains the name of the distribution, and the `par` argument specifies, as a dictionary, the distribution parameters. In the latter, each item key–value pair represents a distribution parameter name and its value. Refer to `distributions` module for a list of the distribution names and their parameter specifications.

The `discretize` method of the `Severity` class produces the discrete severity probability sequence according to the approaches described above. Below, we provide an example for mass dispersal.

```
>>> massdispersal = severity.discretize(
    discr_method='massdispersal',
    n_discr_nodes=50000,
    discr_step=.01,
    deductible=0
    )
```

In order to perform the discretization, the following arguments are needed:

- The chosen discretization method via discr_method.
- The number of nodes (m) set in the n_discr_nodes argument.
- The severity discretization step (h) is in the discr_step argument.
- If necessary, a deductible specifying where the discretization begins. The default value is zero.

After the discretization is achieved, the mean of the discretized distribution can be calculated.

```
>>> import numpy as np
>>> discrete_mean = np.sum(massdispersal['nodes'] * massdispersal['fj'])
>>> print('Mean of the discretised distribution:', discrete_mean)
Mean of the discretised distribution: 5.000000000000079
```

Additionally, the arithmetic distribution obtained via the severity discretization can be visually examined using the plot_discretized_severity_cdf. This method is based on the pyplot interface to matplotlib (Hunter, 2007). Hence, plot_discretized_severity_cdf can be used together with pyplot functions and can receive pyplot.plot arguments to change its output. In the following code blocks, we adopt the plot_discretized_severity_cdf method in conjunction with the plot function from matplotlib.pyplot to compare the cdf of a gamma distribution with mean and variance equal to 5, with the arithmetic distribution obtained with the method of mass dispersal above. We first import the gamma distribution from the distributions module and compute the true cdf.

```
>>> from gemact import distributions
>>> dist = distributions.Gamma(a=5)
>>> nodes = np.arange(0, 20)
>>> true_cdf = dist.cdf(nodes)
```

Next, we plot the discrete severity using the plot_discr_sev_cdf method.

```
>>> import matplotlib.pyplot as plt
>>> severity.plot_discr_sev_cdf(
    discr_method='massdispersal',
    n_discr_nodes=20,
    discr_step=1,
    deductible=0,
    color='#a71429'
    )
>>> plt.plot(nodes, true_cdf, color='#4169E1')
>>> plt.title('Method of Mass Dispersal')
>>> plt.xlabel('z')
>>> plt.show()
```

The arguments discr_method, n_discr_nodes, discr_step, and deductible can be used in the same manner as those described in the discretize method. The argument color is from the matplotlib.pyplot.plot method. The methods title and xlabel were also exported from matplotlib.pyplot.plot to add custom labels for the title and the x-axis (Hunter, 2007).

The output of the previous code is shown in the top left graph of Fig. 1. To obtain the other graphs, simply set discr_method to the desired approach, that is, 'upper_discretisation,' 'lower_discretisation', or 'localmoments.'

3.4 Supported distributions

The gemact package makes for the first time the $(a, b, 0)$ and $(a, b, 1)$ distribution classes (Klugman *et al.*, 2012, p. 81) available in Python. In the following code block, we show how to use our implementation of the zero-truncated Poisson from the `distributions` module.

```
>>> ztpois = distributions.ZTPoisson(mu=2)
```

Each distribution supported in gemact has various methods and can be used in a similar fashion to any scipy distribution. Next, we show how to compute the approximated mean via MC simulation, with the random generator method for the `ZTPoisson` class.

```
>>> random_variates = ztpois.rvs(10**5, random_state=1)
>>> print('Simulated Mean: ', np.mean(random_variates))
Simulated Mean: 2.3095
>>> print('Exact Mean: ', ztpois.mean())
Exact Mean: 2.3130352854993315
```

Furthermore, supported copula functions can be accessed via the `copulas` module. Below, we compute the cdf of a two-dimensional Gumbel copula.

```
>>> from gemact import copulas
>>> gumbel_copula = copulas.GumbelCopula(par=1.2, dim=2)
>>> values = np.array([ [.5, .5]])
>>> print(' Gumbel copula cdf: ', gumbel_copula.cdf(values)[0])
Gumbel copula cdf: 0.2908208406483879
```

In the above example, it is noted that the copula parameter and dimension are defined by means of the `par` and the `dim` arguments, respectively. The argument of the `cdf` method must be a numpy array whose dimensions meet the following requirements. Its first component is the number of points where the function shall be evaluated and its second component equals the copula dimension (`values.shape` of the example is in fact `(1,2)`).

The complete list of the distributions and copulas supported by gemact is available in Section A. We remark that the implementation of some distributions is available in both gemact and scipy.stats. However, the objects of the `distributions` module include additional methods that are specific to their use in actuarial science. Examples are `lev` and `censored_moment` methods, which allow the calculation of the limited expected value and censored moments of continuous distributions. Furthermore, the choice of providing a `Severity` class is in order to have a dedicated object that includes functionalities relevant only for the calculation of a loss model and not for distribution modeling in general. An example of this is the `discretize` method. A similar reasoning applies to the `Frequency` class.

3.5 Illustration `lossmodel`

The following are examples of how to get started and use `lossmodel` module and its classes for costing purposes. As an overview, Fig. 2 schematizes the class diagram of the `lossmodel` module, highlighting its structure and the dependencies of the `LossModel` class.

The `Frequency` and the `Severity` classes represent, respectively, the frequency and the severity components of a loss model. In these, `dist` identifies the name of the distribution and `par` specifies its parameters as a dictionary, in which each item key–value pair corresponds to a distribution parameter name and value. Please refer to `distributions` module for the full list of the distribution names and their parameter specifications. The code block below shows how to initiate a frequency model.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

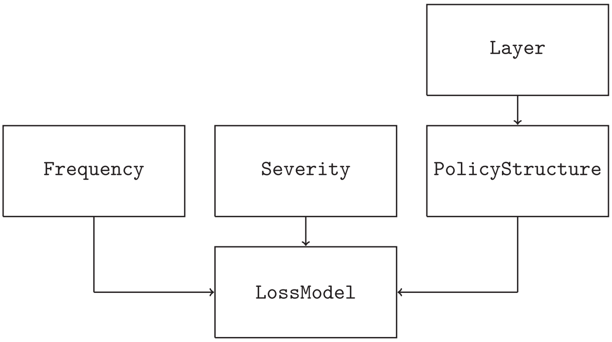


Figure 2. Class diagram of the `lossmodel` module. A rectangle represents a class; an arrow connecting two classes indicates that the target class employs the origin class as an attribute. In this case, a `LossModel` object entails `Frequency`, `Severity`, and `PolicyStructure` class instances. These correspond to the frequency model, the severity model, and the policy structure, respectively. The latter, in particular, is in turn specified via one or more `Layer` objects, which include coverage modifiers of each separate policy component.

```
>>> from gemact.lossmodel import Frequency
>>> frequency = Frequency(
    dist='poisson',
    par={'mu': 4},
    threshold=0
    )
```

In practice, losses are reported only above a certain threshold (the reporting threshold) and the frequency model can be estimated only above another, higher threshold called the analysis threshold (Parodi, 2014, p. 323). This can be specified in `Frequency` with the optional parameter `threshold`, whose default value is 0 (i.e., the analysis threshold equals the reporting threshold). Severity models in gemact always refer to the reporting threshold.

A loss model is defined and computed through the `LossModel` class. Specifically, `Frequency` and `Severity` objects are assigned to `frequency` and `severity` arguments of `LossModel` to set the parametric assumptions of the frequency and the severity components. Below, we use the `severity` object we instanced in Subsection 3.3.

```
>>> from gemact.lossmodel import LossModel
>>> lm_mc = LossModel(
    frequency=frequency,
    severity=severity,
    aggr_loss_dist_method='mc',
    n_sim=10**5,
    random_state=1
    )
INFO:lossmodel|Approximating aggregate loss distribution via Monte Carlo
simulation
INFO:lossmodel|MC simulation completed
```

In the previous example, in more detail, `lm_mc` object adopts the MC simulation for the calculation of the aggregate loss distribution. This approach is set via the `aggr_loss_dist_method` equal to '`mc`.' The additional parameters required for the simulation are as follows:

- the number of simulations `n_sim`,
- the (pseudo)random number generator initializer `random_state`.

The cdf of the aggregate loss distribution can be displayed with the `plot_dist_cdf` method. Moreover, a recap of the computation specifications can be printed with the `print_aggr_loss_specs` method.

```
>>> lm_mc.print_aggr_loss_method_specs()
Aggregate Loss Distribution: layer 1
=========================================================
                    Quantity                     Value
=========================================================
 Aggregate loss dist. method                        mc
        Number of simulation                    100000
                Random state                         1
```

The aggregate loss mean, standard deviation, and skewness can be accessed with the `mean`, `std`, and `skewness` methods, respectively. The code below shows how to use these methods.

```
>>> lm_mc.mean(use_dist=True)
19.963155575580227
>>> lm_mc.mean(use_dist=False)
20.0
>>> lm_mc.coeff_variation(use_dist=True)
0.5496773171375182
>>> lm_mc.coeff_variation(use_dist=False)
0.5477225575051661
>>> lm_mc.skewness(use_dist=True)
0.6410913579225725
>>> lm_mc.skewness(use_dist=False)
0.6390096504226938
```

When the `use_dist` argument is set to `True` (default value), the quantity is derived from the approximated aggregate loss distribution (`dist` property). Conversely, when it is `False`, the calculation relies on the closed-form formulas of the moments of the aggregate loss random variable. These can be obtained directly from the closed-form moments of the frequency and the severity model transformed according to the coverage modifiers (Parodi, 2014, p. 322). This option is available for `mean`, `std`, `var` (i.e., the variance), `coeff_variation` (i.e., the coefficient of variation), and `skewness` methods. It should be noted that the calculation with `use_dist=False` is not viable when aggregate coverage modifiers are present. In such circumstance, the method must necessarily be based on the approximated aggregate loss distribution. In any situations, it is possible to get the moments of the approximated aggregate loss distribution via the `moment` method. The `central` and `n` arguments specify, respectively, whether the moment is central and the order of the moment.

```
>>> lm_mc.moment(central=False, n=1)
19.963155575580227
```

Furthermore, for the aggregate loss distribution, the user can simulate random variates via the `rvs` method. The quantile and the cdf functions can be computed via the `ppf` and the `cdf` methods. Below is an example of the `ppf` method returning the 0.80- and 0.70-level quantiles.

```
>>> lm_mc.ppf(q=[0.80, 0.70])
array([28.81343738, 24.85497983])
```

The following code block shows the costing of a "20 excess 5" XL reinsurance contract. Coverage modifiers are set in a `PolicyStructure` object.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

```
>>> from gemact.lossmodel import PolicyStructure, Layer
>>> policystructure = PolicyStructure(
    layers=Layer(
    cover = 20,
    deductible = 5
    ))
```

More precisely, in the Layer class, the contract cover and the deductible are provided. Once the model assumptions are set and the policy structure is specified, the aggregate loss distribution can be computed.

```
>>> lm_XL = LossModel(
    frequency=frequency,
    severity=severity,
    policystructure=policystructure,
    aggr_loss_dist_method='fft',
    sev_discr_method='massdispersal',
    n_aggr_dist_nodes = 2**17
    )
INFO:lossmodel|Approximating aggregate loss distribution via FFT
INFO:lossmodel|FFT completed
```

It can be noted that, in the previous code block, we determined the aggregate loss distribution with 'fft' as aggr_loss_dist_method. In such case, LossModel requires additional arguments for defining the computation process, namely:

- The number of nodes of the aggregate loss distribution n_aggr_dist_nodes.
- The method to discretize the severity distribution sev_discr_method. Above, we opted for the method of mass dispersal ("massdispersal").
- The number of nodes of the discretized severity n_sev_discr_nodes (optional).
- The discretization step sev_discr_step (optional). When a cover is present, **gemact** automatically adjusts the discretization step parameter to have the correct number of nodes in the transformed severity support.

The same arguments shall be specified while computing the aggregate loss distribution with the recursive formula, that is, aggr_loss_dist_method set to "recursive."

The costing specifications of a LossModel object can be accessed with the method print_costing_specs().

```
>>> lm_XL.print_costing_specs()
                               Costing Summary: Layer 1
===========================================================================
                                               Quantity       Value
===========================================================================
                                                  Cover        20.0
                                             Deductible         5.0
                                        Aggregate cover         inf
                                   Aggregate deductible           0
        Pure premium (dist est.) before share partecip.        3.51
                    Pure premium before share partecip.        3.51
                                        Share partecip.           1
                               Pure premium (dist est.)        3.51
                                           Pure premium        3.51
```

The previous output exhibits a summary of the contract structure (cover, deductible, aggregate cover, and aggregate deductible) and details about the costing results. It is noted that the default value of the share participation equals 1, that is, a is equal to 1 in Equation (2).

Similar to the previous example, user can access moments of the aggregate loss calculated from the approximated distribution and from the closed-from solution.

```
>>> lm_XL.mean(use_dist=True)
3.509346100359707
>>> lm_XL.mean(use_dist=False)
3.50934614394912
>>> lm_XL.coeff_variation(use_dist=True)
1.0001481880266856
>>> lm_XL.coeff_variation(use_dist=False)
1.0001481667319252
>>> lm_XL.skewness(use_dist=True)
1.3814094240544392
>>> lm_XL.skewness(use_dist=False)
1.3814094309741256
```

The next example illustrates the costing of an XL with RS. The `PolicyStructure` object is set as follows.

```
>>> policystructure_RS = PolicyStructure(
    layers=Layer(
    cover = 100,
    deductible = 0,
    aggr_deductible = 100,
    reinst_percentage = 1,
    n_reinst = 2
    ))
```

The relevant parameters are as follows:

- the aggregate deductible parameter `aggr_deductible`,
- the number of reinstatements `n_reinst`,
- the reinstatement percentage `reinst_percentage`.

Below, we compute the pure premium of Equation (7), given the parametric assumptions on the frequency and the severity of the loss model.

```
>>> lm_RS = LossModel(
    frequency=Frequency(
    dist='poisson',
    par={'mu': .5}
    ),
    severity=Severity(
    dist='pareto2',
    par={'scale': 100, 'shape': 1.2}
    ),
    policystructure = policystructure_RS,
    aggr_loss_dist_method='fft',
    sev_discr_method='massdispersal',
    n_aggr_dist_nodes = 2**17
    )
```

```
>>> print(Pure premium (RS):, lm_RS.pure_premium_dist[0])
Pure premium (RS): 4.319350355177216
```

A PolicyStructure object can handle multiple independent layers simultaneously. These can overlap and do not need to be contiguous. The length property indicates the number of layers. The code block below deals with three Layer objects, the first without aggregate coverage modifiers and (participation) share of 0.5, the second with reinstatements, and the third with an aggregate cover.

```
>>> policystructure=PolicyStructure(
    layers=[
    Layer(cover = 100, deductible = 100, share = 0.5),
    Layer(cover = 200, deductible = 100, n_reinst = 2,
    reinst_percentage = 0.6),
    Layer(cover = 100, deductible = 100, aggr_cover = 200)
    ])
>>> lossmodel_multiple = LossModel(
    frequency=Frequency(
    dist='poisson',
    par={'mu': .5}
    ),
    severity=Severity(
    dist='genpareto',
    par={'loc': 0, 'scale': 83.34, 'c': 0.834}
    ),
    policystructure=policystructure
    )
WARNING:lossmodel|Aggregate loss distribution calculation is omitted as
                   aggr_loss_dist_method is missing
WARNING:lossmodel|Layer 2: costing is omitted as aggr_loss_dist_method is
missing
WARNING:lossmodel|Layer 3: costing is omitted as aggr_loss_dist_method is
missing
```

As outlined by the warning messages, since the instantiation of lossmodel_multiple lacks of aggr_loss_dist_method, the calculation of the aggregate loss distribution is omitted. Therefore, costing results are accessible solely for the first Layer, since this is the only one without aggregate coverage modifiers. This fact is reflected in pure_premium and pure_premium_dist properties, containing the premiums derived from the closed-form means and the approximated aggregate loss distribution means, respectively. The latter are indeed not available.

```
>>> lossmodel_multiple.pure_premium
[8.479087307840043, None, None]
>>> lossmodel_multiple.pure_premium_dist
[None, None, None]
```

Contrarily, once the aggregate loss distribution is determined (via dist_calculate method), all layer premiums in pure_premium_dist are available from the costing (costing method). As expected, pure_premium content remains unaffected.

```
>>> lossmodel_multiple.dist_calculate(
    aggr_loss_dist_method='fft',
    sev_discr_method='massdispersal',
    n_aggr_dist_nodes = 2**17
    )
```

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

```
INFO:lossmodel|Computation of layers started
INFO:lossmodel|Computing layer: 1
INFO:lossmodel|Approximating aggregate loss distribution via FFT
INFO:lossmodel|FFT completed
INFO:lossmodel|Computing layer: 2
INFO:lossmodel|Approximating aggregate loss distribution via FFT
INFO:lossmodel|FFT completed
INFO:lossmodel|Computing layer: 3
INFO:lossmodel|Approximating aggregate loss distribution via FFT
INFO:lossmodel|FFT completed
INFO:lossmodel|Computation of layers completed
>>> lossmodel_multiple.costing()
>>> lossmodel_multiple.pure_premium_dist
[8.479087307062226, 25.99131088702302, 16.88704720494799]
>>>lossmodel_multiple.pure_premium
[8.479087307840043, None, None]
```

At last, it is remarked that each `Layer` in `PolicyStructure` is associated with an index `idx`, starting from 0, based on the layer order of the instantiation of the `PolicyStructure` object. This is of help when the user needs to retrieve particular information and features, or to apply methods to one specific layer. All the methods that include the `idx` argument have 0 as default value, meaning that they are applied to the first (or only) layer unless otherwise specified. For example, the `print_policy_layer_specs` method produces a table of recap of the features of the layer indicated by the `idx` argument. Below `idx` equals 1, namely the second Layer in `policystructure`.

```
>>> lossmodel_multiple.print_policy_layer_specs(idx=1)
     Policy Structure Summary: layer 2
=========================================
              Specification          Value
=========================================
                 Deductible          100.0
                      Cover          200.0
       Aggregate deductible              0
       Reinstatements (no.)              2
 Reinst. layer percentage 1            0.6
 Reinst. layer percentage 2            0.6
        Share partecipation              1
```

Likewise, the code block below returns the aggregate loss distribution mean of the third `Layer`, as `idx` is set to 2.

```
>>> lossmodel_multiple.mean(idx=2)
16.88704720494799
```

3.6 Comparison of the methods for computing the aggregate loss distribution

In this section, we analyze accuracy and speed of the computation of the aggregate loss distribution using FFT, recursive formula (recursion), and MC simulation approaches, as the number of nodes, the discretization step, and the number of simulations vary.

For this purpose, a costing example was chosen such that the analytical solutions of the moments of the aggregate loss distributions are known and can be compared to those obtained

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

from the approximated aggregate loss distribution. The values of the parameters of the severity and frequency models are taken from the illustration in Parodi (2014, p. 262). Specifically, these and the policy structure specifications are as follows.

- Severity: lognormal distribution with parameters shape $=1.3$ and scale $=36315.49$, hence whose mean and standard deviation are 84541.68 and 177728.30, respectively.
- Frequency: Poisson distribution with parameter $\mu=3$, with analysis threshold d. It belongs to the $(a, b, 0)$ family described in Subsection 3.2.2 with parameters $a=0$, $b=3$ and $p_0=e^{-3}$.
- Policy structure: contract with deductible $d=10000$.

In particular, the accuracy in the approximation of the aggregate loss distribution has been assessed using the relative error in the estimate of the mean, coefficient of variation (CoV), and skewness, with respect to their reference values (i.e., error = estimate/reference - 1). The latter are obtained using the following closed-form expressions (Bean, 2000, p. 382):

$$\mathbb{E}\left[X\right]=\mu\mathbb{E}\left[L_{d,\infty}(Z)\right],$$

$$\mathrm{CoV}\left[X\right]=\frac{\mathbb{E}\left[L_{d,\infty}(Z)^2\right]^{1/2}}{\mu^{1/2}\mathbb{E}\left[L_{d,\infty}(Z)\right]},$$

$$\mathrm{Skewness}\left[X\right]=\frac{\mathbb{E}\left[L_{d,\infty}(Z)^3\right]}{\mu^{1/2}\mathbb{E}\left[L_{d,\infty}(Z)^2\right]^{3/2}}.$$

Their values for the example are in the top part of Table 1. The test of the speed of our implementation has been carried out by measuring the execution time of the approximation of the aggregate loss distribution function with the built-in `timeit` library. In line with best practice (Martelli *et al.*, 2005, Chapter 18), the observed minimum execution time of independent repetitions of the function call was adopted.

The results of the analysis are reported in Table 1. We considered for FFT and recursion the values 50, 100, 200, and 400 for the discretization step h, and 2^{14}, 2^{16}, 2^{18} for the number of nodes. It can be noted that FFT and recursion produce similar figures in terms of accuracy, but the former is drastically faster. This is expected as FFT takes essentially $\mathcal{O}(m\log(m))$ operations, compared to the $\mathcal{O}(m^2)$ operations for recursion (Embrechts & Frei, 2009). Furthermore, for this example, in both FFT and recursion, when h increases, the error reduces. Finally, MC approach lies in between the two other alternatives, when it comes to computing time.

3.7 Comparison with aggregate FFT implementation

The aggregate package (Mildenhall, 2022) allows to compute the aggregate loss distribution using FFT. In this section, we compare our implementation with aggregate (version `0.9.3`) implementation to show that the two provide similar results. We adopt the same underlying frequency and severity assumptions of Subsection 3.6 and contracts with different combinations of individual and aggregate coverage modifiers. In particular, we first consider no reinsurance, then an XL, with individual-only coverage modifiers, a SL, and finally an XL with individual and aggregate coverage modifiers (XL w/agg.). In line with the example in Parodi (2014, p. 262), individual coverage modifiers are $c=1000000$ and $d=10000$, and aggregate coverage modifiers are $u=1000000$ and $v=50000$. The number of nodes m is set to 2^{22} in all the calculations.

The comparison of the speed of the two implementations has been carried out by means of the built-in `timeit` library. In particular, we measured the execution time of both the initialization of the main computational object and the calculation of the aggregate loss distribution in order to make the comparison consistent and adequate. In line with best practice (Martelli *et al.*, 2005,

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Table 1. Accuracy and speed of the approximation of the aggregate loss distribution using fast Fourier transform (FFT), the recursive formula (recursion), and the Monte Carlo (MC) simulation when varying the number of nodes (m), the discretization step (h), and number of simulations. The upper table contains the reference values obtained from the closed-form solutions. The lower table reports the execution times in second and the relative errors with respect to the reference values

	Mean	CoV	Skewness
Reference values	268837	1.35520	7.02399

Method	Time (sec.)	Mean	CoV	Skewness
FFT ($h = 50, m = 2^{**}14$)	0.002	−2.76451e-01	−3.00646e-01	−8.06071e-01
FFT ($h = 100, m = 2^{**}14$)	0.002	−8.96804e-02	−1.98006e-01	−7.15193e-01
FFT ($h = 200, m = 2^{**}14$)	0.003	−2.19140e-02	−1.01312e-01	−5.69628e-01
FFT ($h = 400, m = 2^{**}14$)	0.003	−4.41036e-03	−5.88141e-02	−3.47025e-01
FFT ($h = 50, m = 2^{**}16$)	0.010	−2.19153e-02	−1.01317e-01	−5.69635e-01
FFT ($h = 100, m = 2^{**}16$)	0.009	−4.40959e-03	−5.88200e-02	−3.47044e-01
FFT ($h = 200, m = 2^{**}16$)	0.008	−7.24988e-04	−1.36155e-02	−2.37709e-01
FFT ($h = 400, m = 2^{**}16$)	0.007	−9.66263e-05	−3.49930e-03	−1.14264e-01
FFT ($h = 50, m = 2^{**}18$)	0.037	−7.24464e-04	−1.36142e-02	−2.37694e-01
FFT ($h = 100, m = 2^{**}18$)	0.035	−9.34546e-05	−3.46164e-03	−1.13558e-01
FFT ($h = 200, m = 2^{**}18$)	0.036	−3.61429e-06	−3.69322e-04	−3.07951e-02
FFT ($h = 400, m = 2^{**}18$)	0.041	−1.73491e-06	−2.80936e-05	−6.37228e-03
Recursion ($h = 50, m = 2^{**}14$)	1.397	−2.76451e-01	−3.00646e-01	−8.06071e-01
Recursion ($h = 100, m = 2^{**}14$)	1.368	−8.96804e-02	−1.98006e-01	−7.15193e-01
Recursion ($h = 200, m = 2^{**}14$)	1.398	−2.19140e-02	−1.01312e-01	−5.69628e-01
Recursion ($h = 400, m = 2^{**}14$)	1.390	−4.41046e-03	−4.16016e-02	−4.00405e-01
Recursion ($h = 50, m = 2^{**}16$)	13.401	−2.19154e-02	−1.01317e-01	−5.69636e-01
Recursion ($h = 100, m = 2^{**}16$)	12.606	−4.41006e-03	−4.16096e-02	−4.00430e-01
Recursion ($h = 200, m = 2^{**}16$)	11.815	−7.25738e-04	−1.36242e-02	−2.37782e-01
Recursion ($h = 400, m = 2^{**}16$)	11.771	−9.62811e-05	−3.49125e-03	−1.14125e-01
Recursion ($h = 50, m = 2^{**}18$)	448.168	−7.25465e-04	−1.36260e-02	−2.37794e-01
Recursion ($h = 100, m = 2^{**}18$)	464.185	−9.47567e-05	−3.49381e-03	−1.14141e-01
Recursion ($h = 200, m = 2^{**}18$)	460.200	−1.00139e-05	−6.92170e-04	−4.30010e-02
Recursion ($h = 400, m = 2^{**}18$)	462.339	−2.41068e-06	−1.03582e-04	−1.25319e-02
MC ($2^{**}14$ sim.)	0.243	−1.37610e-02	−3.67078e-02	−3.77663e-01
MC ($2^{**}16$ sim.)	0.985	−5.53096e-03	−2.44089e-02	−3.49520e-01
MC ($2^{**}18$ sim.)	3.922	−3.02675e-03	−6.31524e-03	−8.93550e-02
MC ($2^{**}20$ sim.)	15.901	−1.20257e-03	−5.21299e-03	−7.49223e-02

Chapter 18), the observed minimum execution time of independent repetitions of the function call was adopted.

As reported by Table 2, the two implementations generate consistent results; their estimates for mean, CoV, and skewness tend to coincide for all contracts and are close to the reference values when these are available. When it comes to computing time, gemact takes a similar time, just under one second, for all the examples considered. For the case without reinsurance and the XL, aggregate performs slightly better. Conversely, for SL and XL w/agg., gemact is more than twice as fast.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Table 2. Comparison for different contracts of aggregate and gemact implementation of the aggregate loss distribution computation via FFT. When there are no aggregate coverage modifiers, reference values are given in addition to estimated ones. For the XL and the XL w/agg. contracts, individual conditions are $c = 1000000$ and $d = 10000$; for the SL and the XL w/agg. contracts, aggregate coverage modifiers are $u = 1000000$ and $v = 50000$. Execution times are expressed in seconds

Contract	Library	Time (sec.)	Mean	CoV	Skewness
No reinsurance	Reference value	–	253625	1.34406	7.28410
	gemact	0.9581	253625	1.34406	7.27745
	aggregate	0.8102	253625	1.34406	7.28346
XL	Reference value	–	256355	1.10772	2.08525
	gemact	0.9384	256355	1.10772	2.08525
	aggregate	0.8445	256354	1.10773	2.08527
SL	Reference value	–	–	–	–
	gemact	0.9168	194143	1.24438	1.73658
	aggregate	2.4254	194143	1.24438	1.73658
XL w/agg.	Reference value	–	–	–	–
	gemact	0.9388	206363	1.22464	1.62335
	aggregate	2.5221	206363	1.22465	1.62336

4. Loss aggregation

In insurance and finance, the study of the sum of dependent random variables is a central topic. A notable example is risk management, where the distribution of the sum of certain risks needs to be approximated and analyzed for solvency purposes (Wilhelmy, 2010). Another application is the pricing of financial and (re)insurance contracts where the payout depends on the aggregation of two or more dependent outcomes (see, e.g., Cummins *et al.*, 1999; Wang, 2013). In this section, in contrast with the collective risk theory in Section 3, we model the sum of a given number of random variables $d > 1$ that are neither independent nor necessarily identically distributed.

More specifically, consider now the random vector:

$$(X_1, \ldots, X_d) : \Omega \to \mathbb{R}^d,$$

whose joint cdf

$$H(x_1, \ldots, x_d) = P[X_1 \leq x_1, \ldots, X_d \leq x_d] \tag{14}$$

is known analytically or can be numerically evaluated in an efficient way. For a real threshold s, the `gemact` package implements the *AEP algorithm* and a *MC simulation* approach to model:

$$P[X_1 + \ldots + X_d \leq s], \tag{15}$$

given a set of parametric assumptions on the one-dimensional marginals $X_1, \ldots, X_d$ and their copula.

More specifically, the AEP algorithm is designed to approximate Equation (14) through a geometric procedure, without relying on simulations or numerical integration of a density. In Section B, a brief description of the algorithm is given. For a complete mathematical treatment of the subject, the reader should refer to Arbenz *et al.* (2011).

4.1 Illustration `lossaggregation`

Below are some examples of how to use the `LossAggregation` class. This belongs to the `lossaggregation` module, whose class diagram is depicted in Fig. 3. The main class is `LossAggregation`, which is the computation object of the random variable sum. This depends

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

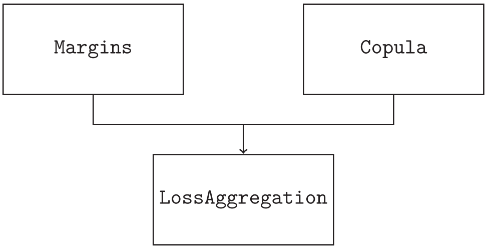


Figure 3. Class diagram of the lossaggregation module. A rectangle represents a class; an arrow connecting two classes indicates that the target class employs the origin class as an attribute. In this case, a LossAggregation object entails Margins and Copula class instances.

on the classes Margins and Copula. Evidently, the former represents the marginal distributions and the latter describes the dependency structure, that is, the copula.

Consistent with the gemact framework, the specifications needed to instantiate Margins and Copula are akin to those of the Frequency and Severity classes in lossmodel. Copula objects are specified by:

- dist: the copula distribution name as a string (str),
- par: the parameters of the copula, as a dictionary.

Likewise, Margins objects are defined by:

- dist: the list of marginal distribution names as strings (str),
- par: the list of parameters of the marginal distributions, each list item is a dictionary.

Please refer to Table A.1 and Table A.2 of Section A for the complete list of the supported distributions and copulas.

```
>>> from gemact import LossAggregation, Copula, Margins
>>> lossaggregation = LossAggregation(
    margins=Margins(
    dist=['genpareto', 'lognormal'],
    par=[{'loc': 0, 'scale': 1/.9, 'c': 1/.9}, {'loc': 0, 'scale': 10,
    'shape': 1.5}],
    ),
    copula=Copula(
    dist='frank',
    par={'par': 1.2, 'dim': 2}
    ),
    n_sim=500000,
    random_state=10,
    n_iter=8
    )
```

Besides marginal and copula assumptions, the instantiation of LossAggregation accepts the arguments n_sim and random_state to set the number of simulation and the (pseudo)random number generator initializer of the MC simulation. If n_sim and the random_state are omitted, the execution is bypassed. Nonetheless, the user can perform it at a later point through the dist_calculate method. Also, n_iter controls the number of iterations of the AEP algorithm. This parameter is optional (default value is 7) and can be specified either at the creation of class or directly in methods that include the use of this approach.

In the following code block, we show how to calculate the cdf of the sum of a generalized Pareto and a lognormal-dependent random variables, using the AEP algorithm ("aep") and the MC simulation approach ("mc"). The underlying dependency structure is a Frank copula.

```
>>> s=300 # arbitrary value
>>> p_aep = lossaggregation.cdf(x=s, method=aep)
>>> print('P(X1+X2 <= s) = ', p_aep)
P(X1+X2 <= s)=0.9811620158197308
>>> p_mc = lossaggregation.cdf(x=s, method='mc')
>>> print('P(X1+X2 <= s) = ', p_mc)
P(X1+X2 <= s)=0.98126
```

LossAggregation includes other functionalities like the survival function sf, the quantile function ppf, and the generator of random variates rvs. Furthermore, for the MC simulation approach, it is possible to derive empirical statistics and moments using methods such as for example moment, mean, var, skewness, and censored_moment. To conclude, the last code block illustrates the calculation of the quantile function via the ppf method.

```
>>> lossaggregation.ppf(q=p_aep, method=aep)
300.0000003207744
>>> lossaggregation.ppf(q=p_mc, method=mc)
299.9929982860278
```

4.2 Comparison of the methods for computing the cdf

In this section, we compared our implementation of the AEP algorithm with the alternative solution based on MC simulation in terms of speed and accuracy. Accuracy in the calculation of the cdf has been assessed by means of the relative error with respect to the reference value for a chosen set of quantiles (i.e., error = estimate/reference - 1). We replicate the experiment in Arbenz *et al.* (2011) and take the reference values that the authors computed in the original manuscript. The analysis considers four different Clayton–Pareto models, for $d = 2, 3, 4, 5$, with the following parametric assumptions. In the two-dimensional case ($d = 2$), the tail parameters γ of the marginal distributions are 0.9 and 1.8; the Clayton copula has parameter $\theta = 1.2$. In three dimensions ($d = 3$), the additional marginal has parameter $\gamma = 2.6$, and the copula has $\theta = 0.4$. For the four-dimensional ($d = 4$) and five-dimensional ($d = 5$) cases, the extra-marginal component has parameter equal to 3.3 and 4, and the Clayton copula has parameter 0.2 and 0.3, respectively. The cdf has been evaluated at the quantiles $s = \{10^0, 10^2, 10^4, 10^6\}$ for $d = 2$ and $d = 3$, and $s = \{10^1, 10^2, 10^3, 10^4\}$, when $d = 4$ and $d = 5$.

The test of the speed of our implementation has been carried out by measuring the execution time of the cdf function for a single quantile with the built-in timeit library. In line with best practice (Martelli *et al.*, 2005, Chapter 18), the observed minimum execution time of independent repetitions of the function call was adopted.

Table 3 shows the results of our comparison of the two methodologies for the calculation of the cdf.

It can be noted that our implementation of the AEP algorithm shows a high accuracy in the calculation of the cdf, for all quantiles and dimensions. Its precision also remains valid for the five-dimensional case. The figures are in line with the results of the original study. In general, in cases considered, the AEP algorithm is closer to the reference values than the MC simulation approach. Nevertheless, the latter shows contained errors whose order of magnitude is 10^{-2} at most, when the number of simulation is set to the lowest value. On the other hand, the AEP algorithm is outperformed by the MC simulation approach in terms of execution speed. The largest gaps have been observed, especially when the number of iterations of the AEP is higher and the dimension is

Table 3. Accuracy and speed of the cdf calculation using the AEP algorithm and the Monte Carlo (MC) simulation approach for the sum of Pareto random variables coupled with a Clayton copula for different dimensions and quantiles. The upper table contains the reference values from Arbenz *et al.* (2011). The lower table reports the execution times in seconds and the relative errors with respect to the reference values. The time column on the left is about the execution times of **gemact**, while the time column on the right (labeled with a *) lists those of the original manuscript

Reference values:					
Quantile	$d=2$	$d=3$	Quantile	$d=4$	$d=5$
$s=10^0$	0.315835041363441	0.190859309689430	$s=10^1$	0.983690398913354	0.983659549676444
$s=10^2$	0.983690398913354	0.983659549676444	$s=10^2$	0.983690398913354	0.983659549676444
$s=10^4$	0.999748719229367	0.999748708770280	$s=10^3$	0.983690398913354	0.983659549676444
$s=10^6$	0.999996018908404	0.999996018515584	$s=10^4$	0.983690398913354	0.983659549676444

Dim.	Method	Time (sec.)	Time* (sec.)	$s=10^0$	$s=10^2$	$s=10^4$	$s=10^6$
$d=2$	AEP (7 iter.)	0.02	0.01	4.62e-11	−1.86e-09	4.13e-08	1.22e-09
	AEP (10 iter.)	0.06	0.06	9.03e-14	5.56e-13	−6.38e-11	3.88e-11
	AEP (13 iter.)	0.95	1.61	6.91e-14	5.04e-13	1.10e-12	4.47e-13
	AEP (16 iter.)	24.0	49.25	−1.72e-13	4.39e-13	1.26e-12	5.66e-13
	MC (10**4 sim.)	0.01	–	−1.00e-02	−2.13e-04	4.87e-05	−3.98e-06
	MC (10**5 sim.)	0.03	–	−8.07e-04	−1.01e-04	1.87e-05	1.60e-05
	MC (10**6 sim.)	0.28	–	−7.90e-05	−3.82e-05	−2.28e-06	1.02e-06
	MC (10**7 sim.)	2.76	–	1.71e-04	6.20e-06	−6.18e-06	−6.81e-07
$d=3$	AEP (7 iter.)	0.04	0.02	−4.61e-06	−1.15e-06	1.12e-06	1.83e-08
	AEP (9 iter.)	0.28	0.41	−1.73e-07	−3.06e-07	2.39e-07	4.26e-09
	AEP (11 iter.)	4.0	6.65	−6.89e-09	−1.13e-08	2.95e-08	7.66e-10
	AEP (13 iter.)	68.4	118.50	−9.12e-10	−1.29e-09	−6.19e-09	−1.09e-10
	MC (10**4 sim.)	0.01	–	5.22e-02	−1.43e-04	−5.13e-05	−3.98e-06
	MC (10**5 sim.)	0.04	–	7.59e-03	−3.56e-04	1.87e-05	−3.98e-06
	MC (10**6 sim.)	0.36	–	−3.87e-03	1.07e-05	−2.29e-06	−1.98e-06
	MC (10**7 sim.)	4.1	–	7.77e-04	4.31e-05	1.61e-06	−4.81e-07
Dim.	Method	Time (sec.)	Time* (sec.)	$s=10^1$	$s=10^2$	$s=10^3$	$s=10^4$
$d=4$	AEP (4 iter.)	0.05	0.03	−1.13e-04	5.03e-04	7.39e-05	9.30e-06
	AEP (5 iter.)	0.38	0.47	−4.45e-04	1.56e-04	2.71e-05	3.42e-06
	AEP (6 iter.)	5.23	7.15	−4.80e-04	−5.09e-05	−3.69e-06	−4.52e-07
	AEP (7 iter.)	83.55	107.70	−4.55e-04	−1.56e-04	−2.26e-05	−2.85e-06
	MC (10**4 sim.)	0.01	–	3.18e-03	−1.92e-03	3.51e-04	4.42e-04
	MC (10**5 sim.)	0.05	–	6.57e-04	−1.71e-04	1.03e-05	−7.74e-06
	MC (10**6 sim.)	0.46	–	−6.00e-04	1.34e-05	−4.75e-06	−2.67e-05
	MC (10**7 sim.)	4.55	–	−2.23e-04	−1.24e-04	−3.29e-05	−5.04e-06
$d=5$	AEP (3 iter.)	0.03	0.01	−4.72e-03	−5.16e-05	5.24e-06	7.22e-07
	AEP (4 iter.)	0.15	0.20	−6.87e-04	3.63e-04	5.30e-05	6.68e-06
	AEP (5 iter.)	2.66	4.37	−1.77e-04	1.94e-04	2.84e-05	3.57e-06
	AEP (6 iter.)	65.61	92.91	−5.14e-13	−9.37e-12	1.44e-11	−1.17e-11
	MC (10**4 sim.)	0.01	–	5.38e-03	−1.67e-03	−3.70e-04	1.40e-04
	MC (10**5 sim.)	0.06	–	−7.25e-04	2.58e-04	−9.29e-06	−9.02e-05
	MC (10**6 sim.)	0.58	–	6.14e-04	−1.89e-04	−1.93e-05	5.81e-06
	MC (10**7 sim.)	5.65	–	5.29e-04	−2.68e-04	−7.70e-05	−1.02e-05

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

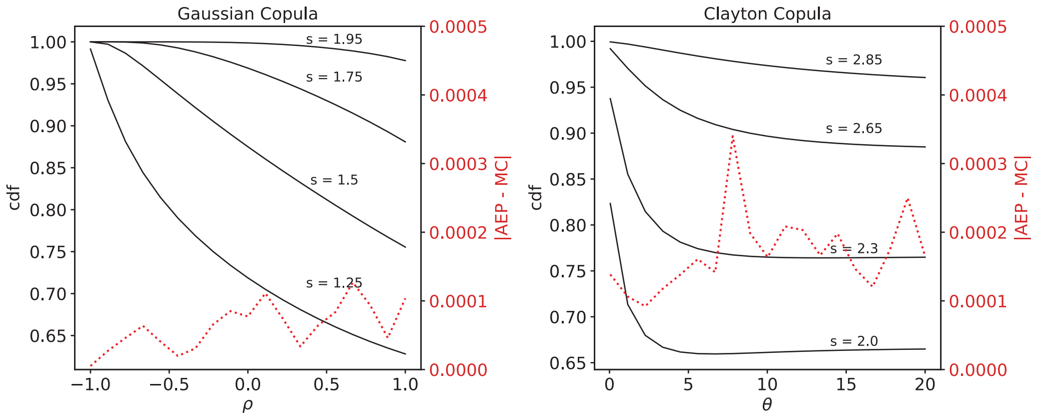


Figure 4. Sensitivity analysis of the cdf at four quantiles s calculated using the AEP algorithm (`n_iter` $= 7$) for different copula models, dimensionality, and underlying degree of dependency. The values of s are 1.25, 1.5, 1.75, and 1.95 for the bivariate Gaussian copula (left plot) and 2, 2.3, 2.65, and 2.85 for the three-dimensional Clayton (right plot). Each solid black line indicates the values of the cdf for a given s, as the respective parameters ρ and θ change. The results of the AEP algorithm correspond to those of the Monte Carlo (MC) simulation approach, using 10^7 number of simulations. The dashed red line represents the average absolute difference between the two method cdf values, calculated across the four quantiles.

4 and 5. However, it should be noted that the computational times for the AEP algorithm remains acceptable, in most cases below one second even in high dimensions. For the sake of completeness and as a reference, the computational time figures of the first implementation, reported in the study of the original manuscript, are also given.

To conclude, we perform a sensitivity study of the two above-mentioned approaches for different dependency structures and number of dimensions. Fig. 4 shows how the cdf value calculated with the AEP algorithm changes as the underlying degree of dependency varies, in the cases of a bivariate Gaussian copula and a three-dimensional Clayton copula. The solid black lines of the graphs represent the cdf evaluated at four quantiles s for increasing values of ρ (the non-diagonal entry of the correlation matrix) and θ parameters, for the Gaussian copula and the Clayton copula, respectively. The cdf values were also compared with those obtained by the MC simulation approach. The average absolute difference between the results of the two methods, across the four quantiles, is highlighted by the dotted red line. It can be seen that this remains stable at low values, regardless of the underlying dependency structure. In all cases, the results produced by the two methods almost coincide.

5. Loss reserve

In non-life insurance, contracts do not settle when insured events occur. At the accident date, the insured event triggers a claim that will generate payments in the future. The task of predicting these liabilities is called *claims reserving* and it assesses claim outstanding loss liabilities (see, e.g., Wüthrich & Merz, 2015, p. 11). In the present work, we refer to the total outstanding loss liabilities of past claims as the *loss reserve* or *claims reserve*. Fig. 5 sketches an example of the timeline evolution of an individual claim. The insured event occurs within the insured period, but the claim settles after several years. In particular, after the claim is reported, the insurance company makes an initial quantification of the claim payment size, the so-called case estimate. Two payments occur thereafter. These payments are not known at the evaluation date, and they require to be estimated. In between the payments, the case estimate is updated. Until a claim settle, the insurance company refers to it as an open claim. In certain circumstances, settled claims can be reopened (Friedland, 2010, p. 431).

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

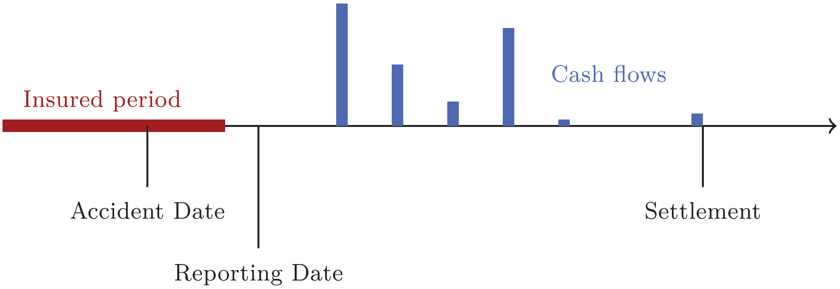


Figure 5. Example of events of a non-life insurance claim.

In this section, we first define the development triangles, the data structure commonly used by actuarial departments for claims reserving (Friedland, 2010, p. 51). These are aggregate representations of the individual claim data. Then, we present the collective risk model for claims reserving in Ricotta & Clemente (2016) and Clemente *et al.* (2019), which allows to estimate the variability of the reserve. This model requires extra-inputs from a deterministic model to be implemented. In this manuscript, we rely on the *Fisher–Lange* model (Fisher & Lange, 1973) as proposed in Savelli & Clemente (2014). Further details can be found in Section C.

5.1 Problem framework

Let the index $i = 0, \ldots, \mathcal{J}$ denotes the claim accident period, and let the index $j = 0, \ldots, \mathcal{J}$ represents the claim development period, over the time horizon $\mathcal{J} > 0$. The so-called *development triangle* is the set:

$$\mathcal{T} = \{(i, j) : i = 0, \ldots, \mathcal{J}, \ j = 0, \ldots, \mathcal{J}; i + j \leq \mathcal{J}\}, \quad \mathcal{J} > 0.$$

Below is the list of the development triangles used in this section.

- The triangle of *incremental paid claims*:
$$X^{(\mathcal{T})} = \left\{x_{i,j} : (i, j) \in \mathcal{T}\right\},$$
with $x_{i,j}$ being the total payments from the insurance company for claims occurred at accident period i and paid in period $i + j$.
- The triangle of the *number of paid claims*:
$$N^{(\mathcal{T})} = \left\{n_{i,j} : (i, j) \in \mathcal{T}\right\},$$
with $n_{i,j}$ being the number of claim payments occurred in accident period i and paid in period $i + j$. The triangle of average claim cost can be derived from the incremental paid claims triangle and the number of paid claims. Indeed, we define
$$m_{i,j} = \frac{x_{i,j}}{n_{i,j}},$$
where $m_{i,j}$ is the average claim cost for accident period i and development period j.
- The triangle of *incremental amounts at reserve*:
$$R^{(\mathcal{T})} = \left\{r_{i,j} : (i, j) \in \mathcal{T}\right\},$$
with $r_{i,j}$ being the amount booked in period $i + j$ for claims occurred in accident period i.
- The triangle of the *number of open claims*:
$$O^{(\mathcal{T})} = \left\{o_{i,j} : (i, j) \in \mathcal{T}\right\},$$
with $o_{i,j}$ being the number of claims that are open in period $i + j$ for claims occurred in accident period i.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Table 4. Parametric assumptions of the CRMR. In the upper table, we show the parameters for $Z_{h;i,j}$, ψ, and q that are gamma-distributed. The parameters of the structure variables are specified from the user starting from the variance of ψ and q, indeed Ricotta and Clemente (2016) assume $\mathbb{E}[\psi] = \mathbb{E}[q] = 1$. The estimator for the average cost of the individual payments is derived with the Fisher–Lange. The variability of the individual payments is instead obtained from the company database. In the lower table, we show the parameter for the claim payment number, that is, a mixed Poisson-gamma distribution with $\hat{n}_{i,j}$ derived from the Fisher–Lange

	Distribution	Quantity	`a`	`scale`
$Z_{h;i,j}$	Gamma	Individual payment cost	$\hat{c}_{Z_{i,j}}^{-2}$	$\hat{\sigma}_{Z_{i,j}}^{2}\hat{m}_{i,j}$
ψ		Structure variable (individual payment cost)	$\hat{\sigma}_{\psi}^{-2}$	$\hat{\sigma}_{\psi}^{2}$
q		Structure variable (payment number)	$\hat{\sigma}_{q}^{-2}$	$\hat{\sigma}_{q}^{2}$

	Distribution	Quantity	`mu`
$N_{i,j}$	Mixed Poisson	Payment number	$\hat{n}_{i,j}q$

- The triangle of the *number of reported claims*:

$$D^{(\mathcal{T})} = \left\{ d_{i,j} : (i,j) \in \mathcal{T} \right\},$$

  with $d_{i,j}$ being the claims reported in period j and belonging to accident period i. Often, the number of reported claims is aggregated by accident period i.

We implemented the model of Ricotta & Clemente (2016) and Clemente *et al.* (2019), hereafter referred to as CRMR, which connects claims reserving with aggregate distributions. Indeed, the claims reserve is the sum of the (future) payments:

$$R = \sum_{i+j>\mathcal{J}} X_{i,j}. \tag{16}$$

The authors represent the incremental payments in each cell of the triangle of incremental payments as a compound mixed Poisson-gamma distribution under the assumptions in Ricotta & Clemente (2016). For $i, j = 0, \ldots, \mathcal{J}$,

$$X_{i,j} = \sum_{h=1}^{N_{i,j}} \psi Z_{h;i,j}, \tag{17}$$

where the index h is referred to the individual claim severity. The random variable ψ follows a gamma distribution, see Table 4, and introduces dependence between claim-sizes of different cells.

5.1.1 Predicting the claims reserve

In order to estimate the value for the claims reserve R, this approach requires additional parametric assumptions on $N_{i,j}$ and $Z_{h;i,j}$ when $i + j > \mathcal{T}$. In this section, we illustrate how to use the results from the Fisher–Lange to determine the parameters of $N_{i,j}$ and $Z_{h;i,j}$. The Fisher–Lange is a deterministic average cost method for claims reserving (Institute and Faculty of Actuaries, 1997, Section H). The interested reader can refer to Section C for a discussion on the Fisher–Lange and an implementation of the model using gemact.

For any cell (i, j), with $i + j > \mathcal{T}$, the Fisher–Lange can be used to determine the expected number of future payments $\hat{n}_{i,j}$ and the future average claim cost $\hat{m}_{i,j}$. The original concept behind the CRMR is found in Savelli & Clemente (2009), where the authors assumed that $N_{i,j}$ is Poisson distributed with mean $\hat{n}_{i,j}$ and $Z_{h;i,j}$ is gamma distributed with mean $\hat{m}_{i,j}$. The CoV of $Z_{h;i,j}$ is

$\hat{c}_{z_j}$, the relative variation of the individual payments in development period j, independent from the accident period. The values for $\hat{c}_{z_j}$ are estimated using the individual claims data available to the insurer at the evaluation date. The CRMR is extended in Ricotta & Clemente (2016) to take into account of the variability of the severity parameter estimation (*Estimation Variance*), in addition to the random fluctuations of the underlying process (*Process Variance*) (Wüthrich & Merz, 2015, p. 28). This is achieved by considering two structure variables (q and ψ), on claim count and average cost, to describe parameter uncertainty on $N_{i,j}$ and $Z_{h;i,j}$. The parametric assumptions of the model are summarized in Table 4. For $i, j = 0, \ldots, \mathcal{J}$, $\hat{m}_{i,j}$ and $\hat{n}_{ij}$ are obtained from the computation of the Fisher–Lange.

5.2 Illustration `lossreserve`

In this section, we show an example of the CRMR using the `lossreserve` module. In this respect, we simulated the claims reserving datasets using the individual claim simulator in Avanzi *et al.* (2021) to generate the upper triangles $X^{(\mathcal{T})}$, $N^{(\mathcal{T})}$, $O^{(\mathcal{T})}$ and $D^{(\mathcal{T})}$. The $R^{(\mathcal{T})}$ upper triangle has been simulated using the simulator of Avanzi *et al.* (2023). The data are simulated using the same assumptions (on an yearly basis) of the *Simple, short tail claims* environment shown in Al-Mudafer *et al.* (2022).

No inflation is assumed to simplify the forecast of the future average costs. Estimating and extrapolating a calendar period effect in claims reserving is a delicate and complex subject, and a more detailed discussions can be found in Kuang *et al.* (2008a, b), Pittarello *et al.* (2023). Furthermore, in practice, insurers might require a specific knowledge of the environment in which the agents operate to set a value for the inflation (Avanzi *et al.*, 2023). Here, we want to limit the assumptions for this synthetic scenario.

Since the entire claim history is available from the simulation and we know the (true) value for the future payments, we can use this information to back-test the performance of our model (Gabrielli & Wüthrich, 2019). In particular, we refer to the actual amount that the insurer will pay in future calendar years as the *actual reserve*, that is, the amount that the insurer should set aside to cover exactly the future obligations. The CRMR is compared with the *chain-ladder* method of Mack (1993), hereafter indicated as CHL, from the R package `ChainLadder` (Gesmann *et al.*, 2022). These data, together with the datasets from Savelli & Clemente (2014), have been saved in the `gemdata` module for reproducibility.

```
>>> from gemact import gemdata
>>> ip = gemdata.incremental_payments_sim
>>> pnb = gemdata.payments_number_sim
>>> cp = gemdata.cased_payments_sim
>>> opn = gemdata.open_number_sim
>>> reported = gemdata.reported_claims_sim
>>> czj = gemdata.czj_sim
```

The `lossreserve` class diagram is illustrated in Fig. 6: the triangular datasets are stored in the `AggregateData` class, and the model parameters are contained in the `ReservingModel` class. The actual computation of the loss reserve is performed with the `LossReserve` class that takes as inputs an `AggregateData` object, a `ReservingModel` object, and the parameters of the claims reserving model computation.

```
>>> from gemact import AggregateData
>>> from gemact import ReservingModel
>>> import numpy as np
>>> ad = AggregateData(
    incremental_payments=ip,
```

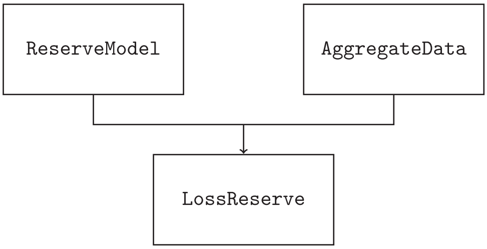


Figure 6. Class diagram of the `lossreserve` module. A rectangle represents a class; an arrow connecting two classes indicates that the target class employs the origin class as an attribute. In this case, a `LossReserve` object entails `ReserveModel` and `AggregateData` class instances.

```
    cased_payments=cp,
    open_claims_number=opn,
    reported_claims=reported,
    payments_number=pnb
    )
>>> resmodel_crm = ReservingModel(
    tail=False,
    reserving_method='crm',
    claims_inflation=np.array([1]),
    mixing_fq_par=.01,
    mixing_sev_par=.01,
    czj=czj
    )
```

In more detail, the `ReservingModel` class arguments are as follows:

- The `tail` parameter (boolean) specifying whether the triangle tail is to be modeled.
- The `reserving_method` parameter, `"crm"` in this case.
- The `claims_inflation` parameter indicating the vector of claims inflation. In this case, as no claims inflation is present, we simply set it to one in all periods.
- The mixing frequency and severity parameters `mixing_fq_par` and `mixing_sev_par`. In Ricotta & Clemente (2016), the authors discuss the calibration of the structure variable. Without having a context of a real-world example, we simply set the structure variables to gamma random variables with mean 1 and standard deviation 0.01, as it is a medium–low risk value in the authors' examples.
- The coefficients of variation of the individual claim severity computed for each development period. In particular, for each development period, the insurer can compute the CoV from the individual observations and save them in the vector `czj`.
- The vector `czj` of the coefficients of variation of the individual claim severity, for each development period. The insurer can compute the CoV from the individual observations, for each development period.

The computation of the loss reserve occurs within the `LossReserve` class:

```
>>> from gemact import LossReserve
>>> lr = LossReserve(
    data=ad,
    reservingmodel=resmodel_crm,
    ntr_sim=1000,
```

Table 5. Reserve and mean squared error of prediction (MSEP) by accident period for the CRMR and the CHL. The actual reserve and its process error (PE) by accident period are also indicated. Amounts are shown in millions

Accident	CRMR		CHL		Actual	
Period	Reserve	MSEP	Reserve	MSEP	Reserve	PE
0	0.00	0.00	0.00	0.00	0.00	0.00
1	404.30	14.37	161.24	0.003	172.03	7.78
2	488.27	15.11	337.34	0.17	327.99	11.29
3	645.25	18.62	532.44	11.58	539.04	17.94
4	795.79	20.34	785.44	27.40	754.93	20.56
5	1026.94	25.16	1134.47	40.79	1090.84	20.73
6	1303.70	29.09	1398.11	52.23	1464.93	28.87
7	1618.36	33.73	1860.56	70.53	1867.04	31.95
8	1963.40	39.51	2214.72	176.97	2382.24	32.70
Total	8246.00	130.09	8424.31	169.79	8599.04	64.32

Table 6. Total reserve estimates, their relative value, as a fraction of the actual value (8599.04), and their coefficients of variation (CoV), for the CRMR and the CHL. Absolute amounts are reported in millions

	Reserve	Reserve/actual	CoV
CRMR	8246.00	0.96	1.58%
CHL	8424.31	0.98	2.02 %

```
    random_state=42
    )
```

To instantiate the `LossReserve` class, `AggregateData` and `ReservingModel` objects need to be passed as arguments. The additional arguments required for the computation are as follows:

- the number of simulated triangles `ntr_sim`,
- the (pseudo)random number generator initializer `random_state`.

The mean reserve estimate for the CRMR can be extracted from the `reserve` attribute. In a similarly way to `LossModel` and `LossAggregation`, the distribution of the reserve can be accessed from the `dist` property. The reserve quantiles can be obtained with the `ppf` method. Output figures are expressed in millions to simplify reading.

```
>>> lr.reserve
8245996481.515498
>>> lr.ppf(q=np.array([.25, .5, .75, .995, .9995]))/10**6
array([8156.79994049, 8244.66099324, 8335.94438221, 8600.42263791,
8676.84206101])
```

5.2.1 Comparison with the chain-ladder

This section compares the CRMR and the CHL with the actual reserve we know from the simulation. For clarity, the reserve of accident period i is defined as $R_i = \sum_{j=\mathcal{T}-i+1}^{\mathcal{T}} X_{i,j}$. Table 5 reports the main results. We can see that the CHL reserve estimate is closer than the CRMR reserve estimate to the actual reserve (the "Reserve" column of each block). In addition to the reserve estimate, we provide the *mean squared error of prediction* (MSEP) (Wüthrich & Merz, 2015, p. 268)

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

of the reserve estimator for each accident period and for the total reserve. In the aforementioned manuscript, the author introduces the MSEP as a measure of risk under the notion in Wüthrich & Merz (2015, p. 169) conditional on the information available in the upper triangle. The MSEP can be decomposed into two components, the irreducible risk arising from the data generation process (process error, PE) and a risk component arising from the model used to calculate the reserve (model error, ME). We can provide a reference value for the true PE by running multiple simulations from the simulator described in the previous section and calculating the standard error of the reserves by accident period. To obtain the PE, we simulated 100 triangles. Net of the ME component, the results in Table 5 appear to show that the CHL underestimates the forecast error in the early accident periods and overestimates the forecast error in the later accident periods.

To conclude, Table 6 shows the total reserves of the CRMR and the CHL. In particular, it highlights the reserve amount as a percentage of the actual reserve (8599.04) and the CoV, that is, the MSEP divided by the reserve. The total relative variability referred to the PE component gives a CoV of 0.76 %.

6. Conclusions

This paper introduces gemact, a Python package for non-life actuarial modeling based on the collective risk theory.

Our library expands the reach of actuarial sciences within the growing community of Python programming language. It provides new functionalities and tools for loss modeling, loss aggregation, and loss reserving, such as the $(a, b, 0)$ and $(a, b, 1)$ classes of distributions, the AEP algorithm and the collective risk model for claims reserving. Hence, it can be applied in different areas of actuarial sciences, including pricing, reserving, and risk management. The package has been designed primarily for the academic environment. Nevertheless, its use as support for insurance business specialists in prototypes modeling, business studies, and analyses is not to be excluded.

The structure of our package aims to ensure modifiability, maintainability, and scalability, as we thought of gemact as an evolving and growing project in terms of introducing features, integrating functionalities, enhancing methodologies, and expanding its scopes. Possible future enhancements could involve the introduction of new probability distribution families, the implementation of supplementary methodologies for the approximation of quantiles of the sum of random variables, and the addition of costing procedures for exotic and nontraditional reinsurance solutions.

Device specifications

All experiments and analyses were run on a computer with an Intel® Core$^{\text{TM}}$ i7-1065G7 CPU processor with 16 GB RAM, running at 1.30 GHz, on Windows 11 Home edition.

Acknowledgments. Previous versions of **gemact** were presented at the Mathematical and Statistical Methods for Actuarial Sciences and Finance 2022, and at the Actuarial Colloquia 2022, in the ASTIN section. We would like to thank all the people who gave us feedback and suggestions about the project.

Data availability statement. The data and code supporting the findings of this study are openly available in GitHub at gpitt71/gemact-code. Supplementary material not included as code blocks in the manuscript can be found in the subfolder vignette. The results contained in the manuscript are reproducible, excluding environment-specific numerical errors. These discrepancies do not affect the overall validity of the results. The gpitt71/gemact-code folder is registered with the unique Zenodo DOI reference number 10.5281/zenodo.10117505.

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

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Competing interests. The author(s) declare none.

References

Avanzi, B., Taylor, G. & Wang, M. (2023). Splice: a synthetic paid loss and incurred cost experience simulator. *Annals of Actuarial Science*, **17** (1), 7–35.

Al-Mudafer, M. T., Avanzi, B., Taylor, G., & Wong, B. (2022). Stochastic loss reserving with mixture density neural networks. *Insurance: Mathematics and Economics*, **105**, 144–174.

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

Antal, P. (2009). *Mathematical methods in reinsurance*. Swiss Re.

Arbenz, P., Embrechts, P., & Puccetti, G. (2011). The aep algorithm for the fast computation of the distribution of the sum of dependent random variables. *Bernoulli*, **17**(2), 562–591.

Avanzi, B., Taylor, G., Wang, M., & Wong, B. (2021). Synthetic: An individual insurance claim simulator with feature control. *Insurance: Mathematics and Economics*, **100**, 296–308.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Bean, M. A. (2000). Probability: the science of uncertainty with applications to investments, insurance, and engineering. Available online at the address https://api.semanticscholar.org/CorpusID:106862438.

Bogaardt, J. (2022). **chainladder** package, version 0.8.13 [Computer software manual]. Available online at the address https://chainladder-python.readthedocs.io/en/latest/intro.html.

Bok, D. (2022). **copulae** package, version 0.7.7 [Computer software manual]. Available online at the address https://copulae.readthedocs.io/en/latest/.

Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance* (pp. 195–203).

Bühlmann, H. (1984). Numerical evaluation of the compound poisson distribution: recursion or fast fourier transform? *Scandinavian Actuarial Journal*, **1984**(2), 116–126.

Clemente, G. P., Savelli, N., & Zappa, D. (2019). Modelling outstanding claims with mixed compound processes in insurance. *International Business Research*, **12**(3), 123–138.

Cummins, D., Lewis, C. & Phillips, R. (1999) Pricing excess-of-loss reinsurance contracts against catastrophic loss. In: **K. A. Froot** (Ed.), *The financing of catastrophe risk* (pp.93–148). University of Chicago Press.

Dutang, C., Goulet, V., & Langevin, N. (2022). Feller-pareto and related distributions: Numerical implementation and actuarial applications. *Journal of Statistical Software*, **103**(6), 1–22.

Dutang, C., Goulet, V., & Pigeon, M. (2008). actuar: An r package for actuarial science. *Journal of Statistical software*, **25**, 1–37.

Embrechts, P., & Frei, M. (2009). Panjer recursion versus fft for compound distributions. *Mathematical Methods of Operations Research*, **69**(3), 497–508.

Fisher, W. H., & Lange, J. T. (1973). Loss reserve testing: a report year approach. *Proceedings of the Casualty Actuarial Society*, **60**, 189–207.

Friedland, J. (2010) Estimating unpaid claims using basic techniques. In *Casualty actuarial society*, vol. **201.**

Gabrielli, A., & Wüthrich, M. V. (2019). Back-testing the chain-ladder method. *Annals of Actuarial Science*, **13**(2), 334–359.

Genz, A., & Bretz, F. (1999). Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts. *Journal of Statistical Computation and Simulation*, **63**(4), 103–117. doi: 10.1080/00949659908811962.

Genz, A., & Bretz, F. (2002). Comparison of methods for the computation of multivariate t probabilities. *Journal of Computational and Graphical Statistics*, **11**(4), 950–971. doi: 10.1198/106186002394. Retrieved from https://doi.org/10.1198/106186002394.

Gerber, H. U. (1982). On the numerical evaluation of the distribution of aggregate claims and its stop-loss premiums. *Insurance: Mathematics and Economics*, **1**(1), 13–18.

Gesmann, M., Murphy, D., Zhang, Y. W., Carrato, A., Wüthrich, M., Concina, F., & Dal Moro, E. (2022). Chainladder: Statistical methods and models for claims reserving in general insurance [Computer software manual]. Received from https://CRAN.R-project.org/package=ChainLadder (R package version 0.2.15).

Grübel, R., & Hermesmeier, R. (1999). Computation of compound distributions i: Aliasing errors and exponential tilting. *ASTIN BULLETIN: The Journal of the IAA*, **29**(2), 197–214.

Hofert, M., & Mächler, M. (2011). Nested archimedean copulas meet R: The nacopula package. *Journal of Statistical Software*, **39**(9), 1–20. Retrieved from https://www.jstatsoft.org/v39/i09/.

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

Institute and Faculty of Actuaries (1997). Claims reserving manual (2nd ed.).

Johansson, N., & Löfgren, A. (2009). Designing for extensibility: An action research study of maximizing extensibility by means of design principles. (B.S. thesis). Univeristy of Gothenburg.

Klugman, S. A., Panjer, H. H., & Willmot, G. E. (2012). *Loss models: From data to decisions*, vol. **715**. John Wiley & Sons.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Kojadinovic, I., & **Yan, J.** (2010). Modeling multivariate distributions with continuous margins using the copula R package. *Journal of Statistical Software*, **34**(9), 1–20. Retrieved from https://www.jstatsoft.org/v34/i09/.

Kuang, D., **Nielsen, B.**, & **Nielsen, J. P.** (2008a). Forecasting with the age-period-cohort model and the extended chain-ladder model. *Biometrika*, **95**(4), 987–991.

Kuang, D., **Nielsen, B.**, & **Nielsen, J. P.** (2008b). Identification of the age-period-cohort model and the extended chain-ladder model. *Biometrika*, **95**(4), 979–986.

Lab, M. D. T. A. (2022). Copulas package, version 0.7.0 [Computer software manual]. Retrieved from https://github.com/sdv-dev/Copulas.

Ladoucette, S. A., & **Teugels, J. L.** (2006). Analysis of risk measures for reinsurance layers. *Insurance: Mathematics and Economics*, **38**(3), 630–639.

Mack, T. (1993). Distribution-free calculation of the standard error of chain ladder reserve estimates. *ASTIN Bulletin: The Journal of the IAA*, **23**(2), 213–225.

Martelli, A., **Ravenscroft, A.**, & **Ascher, D.** (2005). *Python cookbook*. O'Reilly Media, Inc.

Mildenhall, S. (2022). Aggregate package. [Computer software manual]. Retrieved from https://aggregate.readthedocs.io/en/latest/.

Nelsen, R. B. (2007). *An introduction to copulas* (2nd ed.). Springer Science & Business Media. doi: 10.1007/0-387-28678-0.

Nielsen, B. (2015). apc: An r package for age-period-cohort analysis. *The R Journal*, **7**(2), 52.

Ozgur, C., **Colliau, T.**, **Rogers, G.**, & **Hughes, Z.** (2022). Matlab vs. python vs. r. *Journal of Data Science*, **15**(3), 355–372. doi: 10.6339/JDS.201707_15(3).0001.

Panjer, H. H. (1981). Recursive evaluation of a family of compound distributions. *ASTIN Bulletin: The Journal of the IAA*, **12**(1), 22–26.

Parodi, P. (2014). *Pricing in general insurance* (1st ed.). CRC Press. doi:10.1201/b17525.

Pittarello, G., **Hiabu, M.**, & **Villegas, A. M.** (2023). Replicating and extending chain-ladder via an age-period-cohort structure on the claim development in a run-off triangle. arXiv preprint arXiv:2301.03858.

R Core Team. (2017). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from https://www.R-project.org/.

Ricotta, A., & **Clemente, G. P.** (2016). An extension of collective risk model for stochastic claim reserving. *Journal of Applied Finance and Banking*, **6**(5), 45.

Savelli, N., & **Clemente, G. P.** (2014). Lezioni di matematica attuariale delle assicurazioni danni. EDUCatt-Ente per il diritto allo studio universitario dell'Università Cattolica. Retrieved from http://hdl.handle.net/10807/67154.

Savelli, N., & Clemente, G. P. (2009). A collective risk model for claims reserve distribution. In *Proceedings of "Convegno di Teoria del Rischio", Campobasso* (pp. 59–88).

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

Sommerville, I. (2011). *Software engineering*. Pearson Education Inc.

Sundt, B. (1990). On excess of loss reinsurance with reinstatements. *Transactions of the ASTIN Colloquium*, **12**, 73.

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

Virtanen, P., **Gommers, R.**, **Oliphant, T. E.**, **Haberland, M.**, **Reddy, T.**, **Cournapeau, D.** . . . **van Mulbregt, P.** (2020). SciPy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, **17**(3), 261–272. doi: 10.1038/s41592-019-0686-2.

Wang, P. (2013). Risk modeling of multi-year, multi-line reinsurance using copulas. *Journal of Insurance Issues*, **36**(1), 58–81, Retrieved from https://ideas.repec.org/a/wri/journl/v36y2013i1p58-81.html.

Wang, S. (1998). Aggregation of correlated risk portfolios: models and algorithms. In *Proceedings of the casualty actuarial society*, vol. 85 (pp. 848–939).

Wilhelmy, L. (2010). Economic capital: Applying theory to practice. In: Presentation at the Canadian Institute of Actuaries (CIA) Annual Meeting June 29-30, Vancouver.

Wüthrich, M. V. (2023). Non life insurance: Mathematics & statistics. *SSRN Electronic Journal*. doi:10.2139/ssrn.2319328.

Wüthrich, M. V., & Merz, M. (2015). *Stochastic Claims Reserving Manual: Advances in Dynamic Modeling*. Swiss Finance Institute Research Paper No. 15-34. doi:10.2139/ssrn.2649057.

Yan, J. (2007). Enjoy the joy of copulas: with a package copula. *Journal of Statistical Software*, **21**(4), 1–21. Retrieved from https://www.jstatsoft.org/v21/i04/.

A. List of the supported distributions

Table A.1 gives an overview of the distributions available in gemact. In particular, the table presents the distribution name (column one) and its name within gemact apparatus (column two). Moreover, it shows the distribution support (column three) and whether the distribution is supported in scipy (column four).

Lastly, the list of available copulas is provided in Table A.2.

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Table A.1. List of distributions supported by **gemact**

Distribution	gemact name	Support	scipy
Binomial	`binom`	discrete	✓
Geometric	`geom`	discrete	✓
Log-Series	`logser`	discrete	✓
Negative Binomial	`nbinom`	discrete	✓
Poisson	`poisson`	discrete	✓
PWC	`pwc`	discrete	✗
Zero-Modified Poisson	`zmpoisson`	discrete	✗
Zero-Modified Binomial	`zmbinom`	discrete	✗
Zero-Modified Geometric	`zmgeom`	discrete	✗
Zero-Modified Log-Series	`zmlogser`	discrete	✗
Zero-Modified Negative Binomial	`zmnbinom`	discrete	✗
Zero-Truncated Binomial	`ztbinom`	discrete	✗
Zero-Truncated Geometric	`ztgeom`	discrete	✗
Zero-Truncated Negative Binomial	`ztnbinom`	discrete	✗
Zero-Truncated Poisson	`ztpoisson`	discrete	✗
Beta	`beta`	continuous	✓
Burr	`burr12`	continuous	✓
Exponential	`exponential`	continuous	✓
Fisk	`fisk`	continuous	✓
Gamma	`gamma`	continuous	✓
Generalized Beta	`genbeta`	continuous	✗
Generalized Pareto	`genpareto`	continuous	✓
Inverse Burr (Dagum)	`dagum`	continuous	✓
Inverse Gamma	`invgamma`	continuous	✓
Inverse Gaussian	`invgauss`	continuous	✓
Inverse Paralogistic	`invparalogistic`	continuous	✗
Inverse Weibull	`invweibull`	continuous	✓
Log Gamma	`loggamma`	continuous	✓
Lognormal	`lognormal`	continuous	✓
Paralogistic	`paralogistic`	continuous	✗
Pareto (One-Parameter)	`pareto1`	continuous	✗
Pareto (Two-Parameter)	`pareto2`	continuous	✓
PWL	`pwl`	continuous	✗
Uniform	`uniform`	continuous	✓
Weibull	`weibull`	continuous	✓

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Table A.2. List of copulas supported by **gemact**

Copula	gemact name	Family
Ali-Mikhail-Haq	`ali-mikhail-haq`	Archimedean
Clayton	`clayton`	Archimedean
Frank	`frank`	Archimedean
Gumbel	`gumbel`	Archimedean
Joe	`joe`	Archimedean
Gaussian	`gaussian`	Elliptical
Student t	`tstudent`	Elliptical
Fréchet–Hoeffding lower bound (W)	`frechet-hoeffding-upper`	Fundamental
Fréchet–Hoeffding upper bound (M)	`frechet-hoeffding-lower`	Fundamental
Independent	`independent`	Fundamental

B. The AEP algorithm

The AEP algorithm is a numerical procedure, based on a geometric approach, to calculate the joint cdf of the sum of dependent random variables. The graphical interpretation of the basic idea of this algorithm is straightforward, especially in the two-dimensional case. Therefore, to facilitate the reader's understanding, in this section we limit ourselves to the case where $d = 2$ and focus only on its first two iterations. The underlying logic can then be extended to $d \geq 2$ and to any number of iterations.

Let us first define, for $b_1, b_2 \in \mathbb{R}$, a simplex as:

$$\mathcal{S}((b_1, b_2), h) = \begin{cases} \left\{x_1, x_2 \in \mathbb{R} : x_1 - b_1 > 0, x_2 - b_2 > 0, \text{ and } \sum_{k=1}^{2} \left(x_k - b_k\right) \leq h\right\} & \text{if } h > 0, \\ \left\{x_1, x_2 \in \mathbb{R} : x_1 - b_1 \leq 0, x_2 - b_2 \leq 0, \text{ and } \sum_{k=1}^{2} \left(x_k - b_k\right) > h\right\} & \text{if } h < 0, \end{cases}$$

and a square as:

$$\mathcal{Q}((b_1, b_2), h) = \begin{cases} (b_1, b_1 + h] \times (b_2, b_2 + h] & \text{if } h > 0, \\ (b_1 + h, b_1] \times (b_2 + h, b_2] & \text{if } h < 0. \end{cases}$$

The H-measure of the square $V_H\left(\mathcal{Q}(b_1, b_2, h)\right)$ is computed as follows (Nelsen, 2007, p. 8):

$$V_H\left(\mathcal{Q}((b_1, b_2), h)\right) = H\left(b_1 + h, b_2 + h\right) - H\left(b_1, b_2 + h\right) - H\left(b_1 + h, b_2\right) + H\left(b_1, b_2\right),$$

where H the joint cdf in Equation (15). In general, the algorithm is based on the observation that a simplex approximated by a square generates three smaller simplexes, each of which can in turn be approximated by a square that generates three new, even smaller simplexes, and so on. By repeating this iterative scheme with an increasing number of iterations, the quality of the approximation improves and the error tends to 0. It can be noted that some simplexes generated by the process lies outside the original simplex. The measure of those needs to be subtracted instead of being added.

Fig. C.1a shows the simplex $\mathcal{S}_1 = \mathcal{S}((0, 0), s)$, where $s \in \mathbb{R}^+$ is the value at which the joint cdf is calculated in Equation (15). For the first iteration (Fig. C.1b), we adopt the square $\mathcal{Q}_1 = \mathcal{Q}((0, 0), \frac{2}{3}s)$. Arbenz *et al.* (2011) explains that the choice of the 2/3 factor provides fastest convergence when $d = 2$. This factor is set automatically in our implementation. Hence, at the end of the first iteration, we have

$$P\left[X_1 + X_2 \leq s\right] \approx P_1\left(s\right),$$

where $P_1(s) = V_H(\mathcal{Q}_1)$. For example, if we consider uniform marginals and a Gaussian copula with correlation 0.7, we would obtain the following.

```
>>> from gemact import Margins, Copula
>>> margins = Margins(
    dist=['uniform', 'uniform'],
    par=[{'a': 0, 'b': 1}, {'a': 0, 'b': 1}]
    )
>>> copula = Copula(
    dist='gaussian',
    par={'corr': [ [1, 0.7], [0.7, 1]]}
    )
>>> la = LossAggregation(
    copula=copula,
    margins=margins
    )
    >>> la.cdf(x=1, n_iter=1, method='aep')
    0.55188934403716
```

In the second iteration of the algorithm, shown in Fig. 7c, we use again the same logic and approximate the simplexes in Fig. 7b:

$$\mathcal{S}_2 = \mathcal{S}\left(\left(\frac{2}{3}s, 0\right), \frac{1}{3}s\right)$$

$$\mathcal{S}_3 = \mathcal{S}\left(\left(\frac{2}{3}s, \frac{2}{3}s\right), \frac{-1}{3}s\right)$$

$$\mathcal{S}_4 = \mathcal{S}\left(\left(0, \frac{2}{3}s\right), \frac{1}{3}s\right)$$

with the squares:

$$\mathcal{Q}_2 = \mathcal{Q}\left(\left(\frac{2}{3}s, 0\right), \frac{1}{3}s\right)$$

$$\mathcal{Q}_3 = \mathcal{Q}\left(\left(\frac{2}{3}s, \frac{2}{3}s\right), \frac{-1}{3}s\right)$$

$$\mathcal{Q}_4 = \mathcal{Q}\left(\left(0, \frac{2}{3}s\right), \frac{1}{3}s\right).$$

Note that this time the H-measure of $\mathcal{Q}_3$ is subtracted. Similarly to the first iterations, the $\frac{1}{3}$ factors are chosen accordingly to the guidelines of the original manuscript to guarantee the fastest convergence. We obtain, at the second iteration:

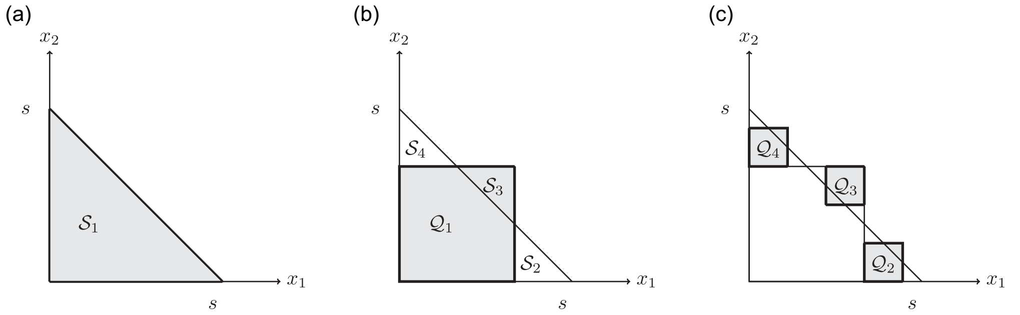
$$P\left[X_1 + X_2 \leq s\right] \approx P_2(s),$$

with $P_2(s) = P_1(s) + V_H(\mathcal{Q}_2) - V_H(\mathcal{Q}_3) + V_H(\mathcal{Q}_4)$.

To conclude, continuing with the previous example, the results for the first two iterations is given in the code block below.

```
>>> la.cdf(x=1, n_iter=2, method='aep')
0.4934418427652146
```

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press



We are interested in $P[X_1 + X_2 \leq s]$. $X_1 + X_2 \leq s$ with $s \in \mathbb{R}^+$ is the simplex $\mathcal{S}_1$.

Iteration 1, the simplex $\mathcal{S}_1$ in the first iteration is approximated with the square $\mathcal{Q}_1$.

Iteration 2, the smaller simplexes $\mathcal{S}_2$, $\mathcal{S}_3$, and $\mathcal{S}_4$ are approximated with the squares $\mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$ and their area is added (or subtracted) to obtain $\mathcal{S}_1$.

Figure C.1. Sketch of the first two iterations of the AEP algorithm in the two-dimensional case.

C. Claims reserving with the Fisher–Lange

This section briefly introduces the Fisher–Lange approach. The claims reserve is the sum of the (future) payments, forecast as the product between the predicted future average cost and the predicted number of future payments, for each cell. In formula:

$$R = \sum_{i+j>\mathcal{J}} \hat{m}_{i,j}\hat{n}_{i,j}. \tag{C1}$$

The average claim cost in the lower triangle is forecast as the projection of the inflated average claim cost:

$$\widehat{m}_{i,j} = m_{\mathcal{J}-j,j} \prod_{h=\mathcal{J}+1}^{i+j} (1+\delta_h), \tag{C2}$$

where δ_h represents the claims inflation for calendar period h.

As far the number of claims are concerned, this method assumes that the future number of paid claims is related to the percentage of open claims at the evaluation date and to the claims settlement speed. Indeed, at the evaluation date, the lower triangle is estimated as:

$$\hat{n}_{i,j} = o_{i,\mathcal{J}-i} \cdot \alpha_{\mathcal{J}-i} \cdot v_j^{(i)}, \tag{C3}$$

where $[\alpha_j]$, $j = 0, \ldots, \mathcal{J}-1$, is the vector of open claims given by $\alpha_j = \mathbb{E}[\tau_{i,j}]$, and

$$\tau_{i,j} = \frac{\sum_{h=j+1}^{\mathcal{J}-i} n_{i,h} + o_{i,\mathcal{J}-i}}{o_{i,j}}, \tag{C4}$$

for $i = 0, \ldots, \mathcal{J}-1$ and $j = 0, \ldots, \mathcal{J}-i-1$. It is assumed that $\alpha_{\mathcal{J}} = 1$. The claim settlement speed is then computed for each accident year. The settlement speed for accident period $\mathcal{J}$ is

$$v_j^{(\mathcal{J})} = \frac{n_{\mathcal{J}-j,j} \cdot \frac{d_{\mathcal{J}}}{d_{\mathcal{J}-j}}}{\sum_{j=1}^{\mathcal{J}} n_{\mathcal{J}-j,j} \cdot \frac{d_{\mathcal{J}}}{d_{\mathcal{J}-j}}}, \tag{C5}$$

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

Table C.1. Reserves by accident period for the CRMR and the Fisher–Lange. We also report the actual reserve. Amounts are shown in millions

	CRMR			
Accident Period	Reserve	MSEP	Fisher–Lange Reserve	Actual Reserve
0	0.00	0.00	0.00	0.00
1	404.30	14.37	404.44	172.03
2	488.27	15.11	488.38	327.99
3	645.25	18.62	645.98	539.04
4	795.79	20.34	795.43	754.93
5	1026.94	25.16	1026.39	1090.84
6	1303.70	29.09	1302.69	1464.93
7	1618.36	33.73	1616.50	1867.04
8	1963.40	39.51	1962.83	2382.24
Total	8246.00	130.09	8242.64	8599.04

where d_i represents the number of reported claims for accident period i, with $i = 0, \ldots, \mathcal{J}$. The formula is corrected for other accident years following the approach in Savelli & Clemente (2014, p. 141).

Similarly to the CRMR described in Section 5, the results for the Fisher–Lange can be computed with the gemact package. Below, we show an example using the simulated datasets from Section 5.

```
>>> from gemact import gemdata
>>> ip = gemdata.incremental_payments_sim
>>> pnb = gemdata.payments_number_sim
>>> cp = gemdata.cased_payments_sim
>>> opn = gemdata.open_number_sim
>>> reported = gemdata.reported_claims_sim
>>> czj = gemdata.czj_sim
>>> claims_inflation = np.array([1])
```

The data are represented in the `AggregateData` class.

```
>>> from gemact.lossreserve import AggregateData
>>> ad = AggregateData(
    incremental_payments=ip,
    cased_payments=cp,
    open_claims_number=opn,
    reported_claims=reported,
    payments_number=pnb)
```

Afterward, we specify the `ReservingModel`. In this example, we fix the parameter `tail` to True to obtain an estimate of the tail.

```
>>> resmodel = ReservingModel(
    tail=False,
    reserving_method='fisher_lange',
    claims_inflation=claims_inflation)
```

Thereafter, the actual computation of the loss reserve is performed within the `LossReserve` class:

https://doi.org/10.1017/S1748499524000022 Published online by Cambridge University Press

```
>>> from gemact.lossreserve import LossReserve
>>> lossreserve = LossReserve(data=ad, reservingmodel=resmodel)
```

The `LossReserve` class comes with a summary view of the estimated reserve per each accident period, in a similar way to the `LossModel` class, the `print_loss_reserve` method. In Table C.1, we report the CRMR results from Table 5 and we add the results for the Fisher–Lange.

As expected, being the Fisher–Lange the underlying methodology to the CRMR, the results for the claims reserve provided from the two approaches are consistent.

On top of this, insights on the behavior of Fisher–Lange $\left[\alpha_j\right]$ and settlement speed $\left[v_j^{(i)}\right]$, for $j = 0, \ldots, \mathcal{J} - 1$, can be inspected with the `plot_alpha_fl` and `plot_ss_fl` methods.

Cite this article: Pittarello G, Luini E and Marchione MM (2024). GEMAct: a Python package for non-life (re)insurance modeling, *Annals of Actuarial Science*, **18**, 342–378. https://doi.org/10.1017/S1748499524000022