Using aggregate

Short introductions to the features of aggregate, one per page

Stephen J. Mildenhall

2026-11-07, built with aggregate 2.1.0a7, commit 78e35db3+

Contents

1 Using `aad` and `agglimit`
Annual aggregate deductible and aggregate limit on an occurrence layer (aad, agglimit) 5

2 Using `AggregateCHF`
Distributions computed by inverting a characteristic function 9

3 Using `AggregateCHF` with stable laws
Accurate stable distributions by tilted inversion, checked against scipy 15

4 Using `AllocationBounds`
Ranges of the natural allocation of a portfolio premium to each unit over all distortions that price the total 21

5 Using `approximate`
Method-of-moments approximations to an aggregate distribution and their error in the tail 25

6 Using `bivariate`
Joint distribution of two aggregates sharing one claim count, coupled by a copula or a discrete bivariate severity 30

7 Using `Bounds`
Pricing bounds: the envelope of all distortions that price a risk to a given premium with given assets 35

8 Using `clash`
Clash covers on two units hit by the same events with the clash statement 40

9 Using deductibles
Franchise and disappearing deductibles in the DecL layers clause 44

10 Using distortions
Distortion functions for risk-adjusted pricing with the DecL distortion statement and the Distortion class 48

11 Using `elt`
Aggregates from event loss tables with the elt clause 53

12 Using `elt` with files
Event loss tables read from CSV and Parquet files with the elt clause 58

13 Using `eventual_ruin` 62
Eventual ruin at a stated margin and capital with Aggregate.eventual_ruin

14 Using explanations 66
The description and explanation strings on aggregate objects

15 Using FFT windows 71
The FFT grid window chosen by the bucket sizer, shown by bs_window_df

16 Using `grossnet`, `grossceded` and `netceded` 75
Joint distribution of gross, ceded and net loss under occurrence reinsurance with the view-pair prefixes

17 Using help 79
Help in aggregate with decl_help, the help method, and parse error reports

18 Using labels 85
Display labels on statements, layers, reinsurance, expenses, and premium with the DecL as clause

19 Using lattice buckets 88
Exact bucket sizes for discrete severities on a dyadic lattice

20 Using loss corridors 95
Loss ratio corridors retained by the cedant on an aggregate quota share with the DecL corridor clause

21 Using `nttagg` 99
Exact compound distributions with the nttagg kind and the number-theoretic transform

22 Using occurrence reinsurance 104
Per-occurrence excess of loss layers, partial placements and towers (occurrence net of, occurrence ceded to)

23 Using `oep` 108
Occurrence exceeding probability and occurrence PML of a Poisson aggregate, compared with aggregate PML (oep)

24 Using `pnl` and `xpnl` 113
Profit and loss statements with the pnl and xpnl kinds and the PnL class

25 Using `pollaczeck_khinchine` and `wiener_hopf` 118
Eventual ruin probability with pollaczeck_khinchine and wiener_hopf

26 Using `price` 122
Distorted ask and bid prices of a loss distribution with Distortion.price and Aggregate.price

27 Using `price` and `evaluate` 126
Premiums from distortions and breakeven distortions from premiums on an Aggregate with price, price_pentagon and evaluate

28 Using `PricingBounds` 130
Exact price ranges of other risks over all distortions that price a reference risk to a given premium

29 Using profit commissions 135
Profit commissions on an aggregate quota share with the DecL pc clause

30 Using recipes
The Underwriter recipe base: the library included with aggregate, the reproductions corpus and your own DecL 139

31 Using reinstatements
Paid reinstatements on an occurrence layer, with a ceded premium that varies with the annual recovery 145

32 Using renewal counts
Renewal claim counts with a years exposure and the wait and dwait clauses 148

33 Using retro rating
Retrospectively rated premium for a large account, set by the account's own loss between a minimum and a maximum 153

34 Using severity parameters
Shape parameters, scale and location, the mean and CV form, and the _ placeholder in a mixture of severity families 156

35 Using `sharpen`
Auditing the FFT grid with sharpen on Aggregate and Portfolio 160

36 Using sliding scale commissions
Sliding scale ceding commissions on an aggregate quota share with the DecL slide clause 164

37 Using `ssev`
Signed severities with ssev, and aggregates or portfolios as severities 168

38 Using `stable`
Stable laws with the stable statement and aggregate.stable.Stable 173

39 Using `stable` names and arithmetic
Named stable and tweedie laws, and arithmetic on stable references 178

40 Using swing rating
Swing-rated aggregate reinsurance, with a ceded premium set by the ceded loss between a minimum and a maximum 183

41 Using tail classes
Tail thickness classes and the FFT grid they size 186

42 Using the structure chart
Reinsurance program diagrams of layers, shares and retentions, with optional Lee curves (the structure chart) 190

43 Using `tsallis`
The Tsallis distortion, a Beta mixture of TVaRs with a closed-form Gini index, and the extended calibration set 195

44 Using `tweedie`
Tweedie parameter conversions and named laws with the tweedie statement 199

45 Using unconditional layers
Conditional and unconditional layer severities, and the trailing ! on a severity 203

46 Using validation
The validation verdict on a computed object, reinsured views, and the warning a failing build emits 206

47 Using `yelt`
Aggregates from year loss tables with the yelt clause 209

48 Using `zm` and `zt`
Zero-truncated and zero-modified claim counts, and the two geometric distributions 213

Using `aad` and `agglimit`

Annual aggregate deductible and aggregate limit on an occurrence layer (aad, agglimit)

Stephen J. Mildenhall

2026-10-30

The recovery from an occurrence layer `y xs a` on the claims $X_1, \dots, X_N$ of a year is $R = \sum_i \min((X_i - a)^+, y)$. An annual aggregate deductible d and an aggregate limit m apply to that annual recovery, and we call them the **annual terms**. Under them the reinsurer pays

$$A(R) = \min((R - d)^+, m)$$

and the cedant keeps $L - A(R)$, where L is the gross annual loss. The amounts are for 100% of the layer: under a share s the reinsurer pays $s\min((R_{100} - d)^+, m) = \min((R - sd)^+, sm)$ on the shared recovery $R = sR_{100}$.

The annual terms are the DecL clauses `aad d` and `agglimit m`[1], written directly after the occurrence layer and before any premium clause. A `pnl` or `xpnl` books them[2]. The reinsurer's side is a function of R alone. The cedant's net is a function of the pair (L, R), not of $L-R$. Neither of L and R determines the other, because the claim count decouples them.

A plain `agg` therefore applies the annual terms after `occurrence ceded to`, as the aggregate stage `aggregate ceded to sm xs sd`, and drops them, with a warning, after `occurrence net of`. A `pnl` statement computes the profit and loss of a program, premium less losses, and an `xpnl` computes the same account in steps from gross to net. Both compute the joint distribution of (L, R) and book the cedant's net.

```
occurrence ceded to 750 xs 250 aad 50
occurrence ceded to 750 xs 250 agglimit 500
occurrence net of 90% po 750 xs 250 aad 50 agglimit 500 deposit 100 cede 10%
```

```
import warnings
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.8.0
[2] From aggregate 1.9.0

The reinsurer's payment under aad and agglimit

The reinsurer's payment $A(R)$ is computed below for an example book with ten claims a year on average, from a lognormal severity with mean 100 and coefficient of variation 2, limited to 1,000. The book cedes a 750 xs 250 occurrence layer under each combination of the annual terms. Each ceded aggregate is the distribution of $A(R)$. The attribute `agg_reins` lists its aggregate stage as (share, limit, attachment).

```
book = 'agg {} 10 claims 1000 xs 0 sev lognorm 100 cv 2 occurrence {} poisson'
ceded = {t: build(book.format('Ceded', f'ceded to 750 xs 250 {t}'))
         for t in ['', 'aad 50', 'agglimit 500', 'aad 50 agglimit 500']}
df = pd.DataFrame.from_dict(
    {t or 'none': [str(c.agg_reins), c.est_m, c.est_sd,
                   c.density_df.p_total.iloc[0]] for t, c in ceded.items()},
    orient='index', columns=['agg_reins', 'mean', 'SD', 'Pr(A = 0)'])
df.index.name = 'terms'
GT(df)
```

terms	agg_reins	mean	SD	Pr(A = 0)
none	None	199.0162	300.7350	0.4173
aad 50	[(1.0, inf, 50.0)]	172.2558	286.0968	0.5087
agglimit 500	[(1.0, 500.0, 0.0)]	152.5279	190.7864	0.4173
aad 50 agglimit 500	[(1.0, 500.0, 50.0)]	133.0354	187.0483	0.5087

The deductible raises the probability of no recovery from 0.417 to 0.509, and the limit cuts the standard deviation (SD) from 301 to 191. The aggregate stage is exactly the map A applied to the annual recovery of the layer. Pushing the distribution of R, the first row of the table, through A with $d = 50$ and $m = 500$ reproduces the mean of the last row, 133.035.

```
plain, both = ceded[''], ceded['aad 50 agglimit 500']
r = plain.density_df
A = np.minimum(np.maximum(r.loss - 50, 0), 500)
print(f'{(A * r.p_total).sum():.6f} {both.est_m:.6f}')
```

```
133.035368 133.035368
```

Under a 90% share the aggregate stage is 450 xs 45 on the shared recovery, *sm* xs *sd* with $s = 0.9$. The reinsurance description lists the two stages in order.

```
c = build(book.format('Share',
                      'ceded to 90% po 750 xs 250 aad 50 agglimit 500'))
print(c.agg_reins)
wprint(c.reins_description)
```

```
[(1.0, 450.0, 45.0)]
Ceded to 90% share of 750 xs 250 per occurrence then ceded to 100% share of 450
xs 45 in the aggregate.
```

The cedant's net under aad and agglimit

A plain `agg` cannot compute the cedant's net under the annual terms. After `occurrence net of` it drops the annual terms (and the deposit) with an `IgnoredDecLClauseWarning` and returns the net

of the plain layer, $L - R$: the gross mean of 952.70 less the layer mean of 199.02. The stored program keeps the full declaration.

```
with warnings.catch_warnings(record=True) as rec:
    warnings.simplefilter('always')
    n = build(book.format(
        'Net', 'net of 750 xs 250 aad 50 agglimit 500 deposit 150'))
gross = n.reins_summary_df.loc[('occ', 'gross', 'agg'), 'EX']
print(rec[0].category.__name__)
print(f'net {n.est_m:.2f} = gross {gross:.2f} less layer {plain.est_m:.2f}')
print(n.pprogram)
```

```
IgnoredDecLClauseWarning
net 753.68 = gross 952.70 less layer 199.02
agg Net
  10 claims
  1000 xs 0
  sev lognorm 100 cv 2
  occurrence net of
    750 xs 250 aad 50 agglimit 500 deposit 150
  poisson
```

A `pnl` or `xpnl` that refers to the stored program as `agg.Net` books the annual terms. The `xpnl` has one step per reinsurance tier, and the step for the layer books its premium and the recovery $A(R)$. The table lists the obligations with the cedant's signs: a loss is negative, a recovery positive.

```
x = build('xpnl NetX 1500 premium less agg.Net')
e = x.economic_df.xs('Obligation', level='Side')[['EX', 'SD']]
e.index = pd.Index(['gross loss', 'layer recovery', 'net loss'], name='leg')
GT(e, float_format=',.2f')
```

leg	EX	SD
gross loss	-952.70	551.50
layer recovery	133.04	187.05
net loss	-819.66	416.48

The recovery, 133.043, agrees with the ceded mean of 133.035 in the first table to the accuracy of the grid of the joint distribution. The net loss, 819.66, is the gross loss less the recovery. The consolidated `pnl` gives the same net margin, 530.34.

```
p = build('pnl NetP 1500 premium less agg.Net')
print(f'recovery {e.iloc[1, 0]:.4f} against {both.est_m:.4f}')
print(f'net margin: xpnl {x.est_m:.4f}, pnl {p.est_m:.4f}')
```

```
recovery 133.0430 against 133.0354
net margin: xpnl 530.3440, pnl 530.3440
```

An aggregate limit of two occurrence limits is the same contract as one free reinstatement: under both the cedant recovers $\min(R, 1500)$. The two accounts agree, with net margin 595.50 and SD 340.35.

```
for layer in ['agglimit 1500 deposit 150', 'deposit 150 reinstatements 1 free']:
    q = build('xpnl R 1500 premium less '
              + book.format('RB', f'net of 750 xs 250 {layer}'))
    print(f'{layer:36s} {q.est_m:.4f} {q.est_sd:.4f}')
```

```
agglimit 1500 deposit 150            595.5028 340.3454
deposit 150 reinstatements 1 free    595.5028 340.3454
```

Refused programs

The annual terms apply to a single occurrence layer, never to an aggregate layer, and not together with reinstatements; the deductible must be non-negative and the limit positive and finite. On `occurrence ceded to` the annual terms become the aggregate stage, so the program takes no `aggregate` clause.

```
pre = 'agg T 10 claims 1000 xs 0 sev lognorm 100 cv 2 '
refuse(*[pre + s for s in [
    'poisson aggregate net of 750 xs 250 aad 50',
    'occurrence net of 750 xs 250 aad 50 and 1000 xs 1000 poisson',
    'occurrence net of 750 xs 250 aad 50 deposit 100 '
    'reinstatements 1 free poisson',
    'occurrence net of 750 xs 250 aad -5 poisson',
    'occurrence net of 750 xs 250 agglimit 0 poisson',
    'occurrence ceded to 750 xs 250 aad 50 poisson '
    'aggregate ceded to 300 xs 0']])
```

```
DecL: 'aad' / 'agglimit' decorate an occurrence layer, not aggregate
reinsurance. An aggregate layer is itself the annual deductible and limit: write
'm xs d'.
DecL: 'aad' / 'agglimit' require a single occurrence layer; found 2. The annual
terms cannot share the occurrence tier with other layers (the engine ceiling is
the 2-D (L, R) joint).
DecL: 'aad' / 'agglimit' cannot combine with 'reinstatements' on the same layer;
both set the layer's annual recovery. Write the annual limit one way or the
other.
DecL: 'aad' must be finite and >= 0, got -5.
DecL: 'agglimit' must be finite and > 0, got 0. Omit the clause for an unlimited
annual recovery.
T: annual terms (aad / agglimit) on an 'occurrence ceded to' program become its
aggregate stage, so the program cannot also carry an 'aggregate' clause. Write
the annual terms into the aggregate clause instead: 'aggregate ceded to inf xs
50' at this share, composed with the existing aggregate layers.
```

Using `AggregateCHF`

Distributions computed by inverting a characteristic function

Stephen J. Mildenhall

2026-10-30

The compound engine behind `agg` discretizes a severity, takes its FFT, applies the count's probability generating function and inverts. Many infinitely divisible distributions[1] have an infinite claim count and no severity in the actuarial sense. They are easier to reach another way: they may have a closed-form characteristic function (chf) $\varphi(t) = \mathsf{P}e^{itX}$. For these the first two steps are unnecessary. Sampling φ on the FFT's frequency grid and inverting with one `irfft` gives the distribution directly. The object `aggregate.characteristic_function.AggregateCHF`[2] with corresponding DecL statement `chfagg`[3] uses this alternative algorithm. It is a peer of `Aggregate`, not a subclass, and its frames, verdict, and members have `Aggregate`'s names and shapes, with the frequency and severity rows empty. It can be built for several families of distributions with closed form chfs. The examples here use three light-tailed families: `norm`, `gamma`, and `invgauss`. The positional numbers are the family's shapes only, scipy's where scipy has the law, and scale and location are arithmetic around it, `c * FAMILY shapes + b`, exactly as on an `agg` severity. A `mean cv` form takes the same arithmetic. The stable family, with heavy tails and tilting, has its own page, Using `AggregateCHF` with stable laws.

```
chfagg G 25 * gamma 4                       # scale and shape
chfagg G gamma 100 cv 0.5 hints{log2=14}    # the mean and cv form, with a grid hint
chfagg N as "Signed" 30 * norm + 100        # no shape; a signed window
chfagg I 400 * invgauss 0.25                # scipy's mu; also invgauss 100 cv 0.5
```

Two facts make the answers checkable. First, the engine returns bucket masses, not density samples: on the grid $x_k = x_0 + kh$, with U uniform on $[-h/2, h/2]$, $\Pr(X \in [x_k - h/2, x_k + h/2)) = hf_{X+U}(x_k)$ and $\varphi_{X+U}(t) = \varphi_X(t)\,\mathrm{sinc}(th/2)$, so inverting $\varphi \cdot \mathrm{sinc}$ gives the masses of the round discretization that `agg` uses, bucket for bucket. Second, the grid is sized from the law's tails: with $K(s) = \log \mathsf{P}e^{sX}$ the cumulant generating function, the Chernoff bound $\Pr(X > x) \le \exp(K(s) - sx)$ gives an edge $x_R = \inf_s(K(s) - \log\varepsilon)/s$ holding mass ε beyond it, and likewise on the left. Two errors remain, both reported: frequencies beyond π/h are dropped, costing about $|\varphi(\pi/h)|$, and mass outside the window wraps onto it.

```
import numpy as np
import scipy.stats as ss
from aggregate import build, qd, __version__
from aggregate.characteristic_function import AggregateCHF
from style import wprint, GT, figures, figure, refuse
figures()
print(__version__)
```

```
1.43.1
```

[1] The gamma, lognormal, Pareto, stable, Poisson, and inverse Gaussian are infinitely divisible, amongst many others.
[2] From aggregate 1.27.0
[3] From 1.30.0

A gamma law checked against scipy

The function `build` takes the DecL statement and returns the object updated. The constructor `AggregateCHF` takes the same law by name, with the full parameters from Python, and the method `update` sizes the grid and computes. The attribute `info` reports the grid chosen: for `25 * gamma 4` the window runs from 0 (the support bound) to about 1,000, with a bucket of 1/64, which is `round_bucket` of the window over 2^{16} points.

```
g = build('chfagg G 25 * gamma 4')
qd(g)
wprint(g.info)
```

```
AggregateCHF object: G. Family 25 * gamma 4. Updated with bucket size
1/64 and log2 = 16. Validation: not unreasonable.

     Mean SD  CV Skew    P01 Median    P99
X
Freq
Sev
Agg   100 50 0.5    1 20.578 91.797 251.12
aggregate object name    G
value_type               loss
claim count              n/a
frequency distribution   n/a
severity distribution    n/a
loss table               n/a
approximate              exact
bs                       1/64
log2                     16
padding                  n/a
sev_calc                 n/a
dsev_bucket              n/a
normalize                False
x_min                    0
x_max                    1,023.98
premium                  n/a
expected loss            100
loss ratio               n/a
P(X=0)                   2.11506e-16
validation_eps           0.0001
reinsurance              none
occurrence reinsurance   no reinsurance
aggregate reinsurance    no reinsurance
validation               not unreasonable
family                   gamma
parameters               25 * gamma 4
support                  0 to inf
scale                    50
total mass               1
outside mass             1.01e-12, below 0, above 1.01e-12
tilt                     (0, 0)
theta                    (0, 0)
edge error               n/a
truncation error         1.57e-15
transfer sigma           n/a
most negative bucket     -6.28e-17
buckets per scale        3.2e+03
checks                   5 ok
frequency tail           none: a characteristic function law
severity tail            none: a characteristic function law
aggregate tail           [0, inf), exponential right tail; not concentrated
```

```
                        (cv=0.5)
bounded                 False
id                      03F5E2DE
```

```
g2 = AggregateCHF('G', 'gamma', shape=4, scale=25)
g2.update()
print(np.abs(g.agg_density - g2.agg_density).max(), g.program, g2.program == '')
```

```
0.0 chfagg G 25 * gamma 4 True
```

The family includes a scipy reference, so the masses can be checked against the exact bucket probabilities $F(x_k + h/2) - F(x_k - h/2)$. They agree to roundoff.

```
h, x = g.bs, g.xs
exact = g.family.reference.cdf(x + h / 2) - g.family.reference.cdf(x - h / 2)
print(f'{np.abs(g.agg_density - exact).max():.2e}')
```

```
6.66e-16
```

The same law through the compound engine is `agg G dfreq [1] sev 25 * gamma 4`, a fixed count of one claim. On a shared grid the two agree bucket for bucket, to about 10^{-11}.

```
a = build('agg G dfreq [1] sev 25 * gamma 4', log2=12, bs=1/4)
g.update(log2=12, bs=1/4)
print(f'{np.abs(a.density_df.p_total - g.density_df.p_total).max():.2e}')
qd(a)
```

```
1.29e-11
Aggregate object: G. Frequency distribution empirical. Severity gamma,
[0, inf), exponential right tail. Updated with bucket size 1/4 and log2
= 12. Validation: not unreasonable.

      Mean  SD  CV Skew  P01 Median    P99
X
Freq     1   0   0
Sev    100  50 0.5    1 20.5  91.75 251.25
Agg    100  50 0.5    1 20.5  91.75 251.25
```

The normal and inverse Gaussian laws

The normal law is supported on the whole line, so its window is signed. The Chernoff edges for the mass budget sit 5 standard deviations either side of the mean; the bucket is rounded up to a power of two, and the width that rounding adds is split between the two sides, so the window runs from -156 to 356, about 8.5 standard deviations each way. The printed percentiles are those of the law, from scipy.

```
n = AggregateCHF('N', 'norm', mean=100, sd=30)
n.update()
print(n.x_min, n.x_max, n.bs)
GT(n.summary_df)
print(ss.norm(100, 30).ppf([0.01, 0.5, 0.99]))
```

```
-156.0 355.9921875 0.0078125
[ 30.20956378 100.         169.79043622]
```

Every family with a mean and coefficient of variation has a `mean cv` form, written with `form='cv'`. The inverse Gaussian with mean 100 and cv 0.5 is the law with shape $\lambda = \mu/\text{cv}^2 = 400$; its scipy reference is `invgauss(mu / lam, scale=lam)`.

```
i = AggregateCHF('I', 'invgauss', form='cv', mean=100, cv=0.5)
i.update()
wprint(i.family, i.family.reference.mean(), i.family.reference.var() ** 0.5 / 100)
GT(i.stats_df.loc['agg', ['mixed', 'empirical', 'error']])
```

```
FamilyInverseGaussian('invgauss 100 cv 0.5') 100.0 0.5
```

measure	mixed	empirical	error
ex1	100.0000	100.0000	-6.73e-09
ex2	12500.0000	12499.9993	-5.86e-08
ex3	1937500.0000	1937499.1939	-4.16e-07
mean	100.0000	100.0000	-6.73e-09
cv	0.5000	0.5000	-1.13e-07
skew	1.5000	1.5000	-2.85e-06

The Tweedie family at every admissible power

A Tweedie law with $1 < p < 2$ is a compound Poisson with gamma severity, and the `tweedie` clause of an `agg` builds it that way (Using `tweedie`). The members with $p > 2$ and $p < 0$ are tilted stable laws with no severity and no count, which only the characteristic function reaches. The DecL statement `chfagg T tweedie p mean dispersion`[4] covers every admissible p (not 1, a lattice law, and not between 0 and 1, where no law exists). For $1 < p < 2$ the atom $e^{-\lambda}$ at 0 is split out of φ before inversion and put back in the bucket at 0, so truncation does not ring around it; the skewness of every member is p times the coefficient of variation.

```
for p in [1.5, 2.5, -1]:
    t = build(f'chfagg T tweedie {p} 100 cv 0.5')
    m, var, skew = t.family.moments
    print(f'p={p:5}  mean {t.est_m:8.4f}  cv {t.est_cv:.5f}  skew {t.est_skew:.4f} = {p} * cv
    atom {t.family.atom}')
```

```
p=  1.5  mean  99.9999  cv 0.50000  skew 0.7499 = 1.5 * cv  atom (0.0, 0.00033546262790251185)
p=  2.5  mean 100.0000  cv 0.50000  skew 1.2500 = 2.5 * cv  atom None
p=   -1  mean 100.0000  cv 0.50000  skew -0.5000 = -1 * cv  atom None
```

The normal inverse Gaussian, variance gamma, and tempered stable laws of finance are infinitely divisible, with a closed-form cumulant function and light tails on both sides. Each has its moments in closed form, and the grid reproduces them.

```
for prog in ['chfagg N 3 * norminvgauss 6 1.5 + 1', 'chfagg V variancegamma 0.2 0.5 -0.1', 'chfagg
C temperedstable 1 5 10 0.5 + 2']:
    a = build(prog)
    m, var, skew = a.family.moments
    print(f'{a.family.name:15} mean {m:8.4f} {a.est_m:8.4f}  sd {var ** 0.5:.5f} {a.est_sd:.5f}
    skew {skew:.4f} {a.est_skew:.4f}')
```

```
norminvgauss    mean   1.7746   1.7746  sd 1.28549 1.28549  skew 0.3112 0.3112
variancegamma   mean  -0.1000  -0.1000  sd 0.21213 0.21213  skew -0.6809 -0.6809
temperedstable  mean   1.7678   1.7678  sd 0.32755 0.32755  skew -0.5570 -0.5570
```

[4] From aggregate 1.31.0

Validation and the truncation error

The table `validation_df` has the same shape as on an `Aggregate`, with the law's moments against the grid's, and the attribute `validation_description` uses the same words. The checks on the inversion itself, the mass outside the window, the truncation error $|\varphi(\pi/h)|$, the most negative bucket, and the buckets per body scale, are lines of `info`, counted in its `checks` line and written out in `validation_explanation`. A clean law has the description `not unreasonable`. An exponential law, `25 * gamma 1`, has a density jump at 0, so $|\varphi(t)|$ decays only like $1/t$: the truncation flag sets, and the ringing shows as a negative bucket and a mean a few parts in 10^4 off. A gamma shape of 3 or more decays fast enough.

```
e = build('chfagg E 25 * gamma 1')
wprint(e.validation_description)
GT(e.validation_df.loc[['Agg']])
print('\n'.join(ln for ln in e.info.splitlines() if ln.startswith(('truncation', 'most negative',
'checks'))))
```

```
fails agg mean, characteristic function truncated at pi / bs, lower bs
truncation error         9.95e-05
most negative bucket     -1.56e-05
checks                   3 ok, 2 flags: most negative bucket; truncation error
```

Choosing the grid of a `chfagg`

Rule A, the default, sizes the window from the tails and the bucket from `log2`. Rule C takes `bs` and `log2` as given, with the left edge from the same rule, and 0 is always a grid point. The method `grid_window` shows the choice without computing. The method `recommended_grid` returns the smallest grid meeting coverage, truncation, and ten buckets per body scale. The table `bs_window_df` lays out the decision in the same columns as on an `Aggregate`, and the method `sharpen` audits it: a sound grid scoring under the target is confirmed and nothing runs, and otherwise the object moves toward the recommended grid, at most one `log2` per call.

```
print(n.grid_window(log2=16))
print(n.grid_window(log2=12, bs=1/8))
print(n.recommended_grid())
GT(n.bs_window_df[['selected', 'x_min', 'x_max', 'bs', 'log2', 'coverage']])
wprint(n.sharpen().sharpen_description)
```

```
(0.0078125, 16, -156.0, (0.0, 0.0))
(0.125, 12, -156.0, (0.0, 0.0))
(8, 2.0)
Sharpen: not run, the grid already scores 0.00116 against a target of 0.5 and
its outside mass is within the budget.
```

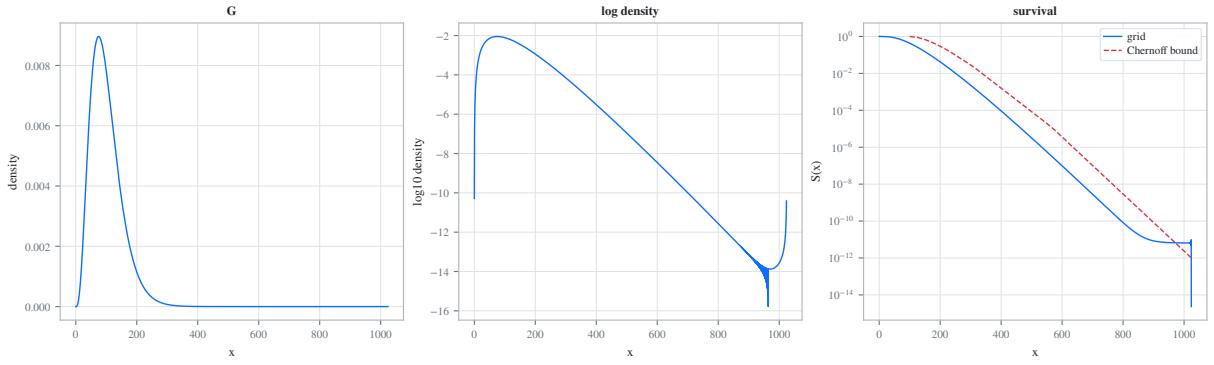
Output tables and the plot of a `chfagg`

The table `density_df` has the same columns as on an `Aggregate`. The table `tail_df` is the return-period ladder, and `tail_behavior_df` gives the tail class on each side. The methods `price`, `evaluate`, and `apply_distortion` work as on an `Aggregate`. The method `plot` shows the density, its logarithm, and the survival function with the Chernoff bound drawn in.

```
GT(g.tail_df.tail(4))
GT(g.tail_behavior_df.loc[['aggregate'], ['min', 'max', 'left_tail', 'right_tail', 'cv']])
```

component	min	max	left_tail	right_tail	cv
aggregate	0	—	bounded	exponential	0.5000

```
fig = g.plot()
```



Refused calls

A number the family does not take is refused with the arithmetic form of the same law. An unknown family, a `mean cv` form on a family without one, a negative multiplier, and a numeric tilt on a law with no power tail are refused too.

```
refuse('chfagg G gamma 4 25', 'chfagg N norm 100 30', 'chfagg X lognorm 1 2',
       'chfagg X stable 1.5 cv 1', 'chfagg X -2 * gamma 4',
       lambda: g.update(tilt=8))
```

```
chfagg: gamma takes 1 shape, shape, got 2. Scale and location are arithmetic
around the law, as on an agg severity: write 'gamma 4 25' as '25 * gamma 4'.
chfagg: norm takes no shape (the standard law), got 2. Scale and location are
arithmetic around the law, as on an agg severity: write 'norm 100 30' as '30 *
norm + 100'.
chfagg: unknown family 'lognorm'; the families are stable, tweedie, norm, gamma,
invgauss, norminvgauss, variancegamma, temperedstable. The statement is 'chfagg
NAME [c *] LAW [+ b | - b], LAW one of stable alpha beta [s0|s1], tweedie p mean
dispersion, tweedie p mean cv c, norm, gamma shape, invgauss mu, norminvgauss a
b, variancegamma sigma nu theta, temperedstable C G M Y, or FAMILY mean cv c'.
DecL parse error at line 1, column 21: Unexpected 'cv'.
chfagg: the scale in 'c * gamma' must be positive, got -2; a reflection is not
supported (for a stable law, negate beta)
G: tilt=8 on a gamma law, which has no power tail; such a law barely wraps, so
there is nothing for a tilt to damp. Use tilt=True or tilt=False.
```

Using `AggregateCHF` with stable laws

Accurate stable distributions by tilted inversion, checked against scipy

Stephen J. Mildenhall

2026-11-01

A stable law with index $\alpha < 2$ has a power tail, $\Pr(X > x) \sim Ax^{-\alpha}$, and no severity to discretize: it is the limit of a Pareto book as the claim count grows (Using `stable`). Its characteristic function is in closed form, so the class `AggregateCHF` (Using `AggregateCHF`) computes it on a grid, accurately into the far tail[1]. In comparison, `scipy.stats.levy_stable` is slow in the body of the distribution and wrong past a few hundred scales. A `chfagg` takes the parameters of a stable law but not a reference such as `stable.x` (Using `stable` names and arithmetic).

The difficulty is aliasing. Inverting on a window of length W folds the mass beyond it back in, and a power tail has a great deal of mass beyond any window: holding the wrapped mass to 10^{-6} untilted takes 2^{23} buckets at $\alpha = 1.1$ and 2^{34} at $\alpha = 0.7$. Worse, the mass from $x + W$ lands on x, so near the right edge the relative error is of order one whatever the total. The cure is to invert along a shifted contour. For $\beta = 1$ the left tail is light and φ is analytic on the upper half plane, so the line of integration moves to $\operatorname{Im} s = \theta$:

$$e^{-\theta x_k} p_k = \frac{1}{2\pi} \int \varphi(t + i\theta)\, h \operatorname{sinc}\big((t + i\theta)h/2\big)\, e^{-itx_k}\, dt.$$

What wraps is the damped sequence $e^{-\theta x} p$, and multiplying back by $e^{\theta x_k}$ recovers the masses. Written as $\tau = \theta W$ e-folds across the window, the wrap costs $\rho e^{-\tau}$ and the amplified roundoff $\varepsilon\kappa e^{\tau}$, both relative to the mass at the right edge; the engine takes the τ^* that balances them and reports the balance as `edge_error`. Tilting halves the number of digits the window's dynamic range costs. The mass beyond the window is no longer folded in: it is left out and reported as `outside_above`, so `sf` stays right across the window and the first bucket holds whatever lies below it.

```
import warnings
import numpy as np
import scipy.stats as ss
from aggregate import build, qd, __version__
from aggregate.characteristic_function import AggregateCHF
from aggregate.constants import ValidationWarning
from aggregate.stable import Stable
from style import wprint, GT, figures, figure
figures()
# some of these laws fail their verdicts on purpose, and the text discusses each
warnings.simplefilter('ignore', ValidationWarning)
print(__version__)
```

```
1.58.0
```

[1] From aggregate 1.28.0

```
chfagg S 2 * stable 1.5 1 + 10            # alpha beta [s0|s1], scaled and shifted, tilted
chfagg W stable 0.7 1 hints{tilt=False}   # untilted, wrapping, for teaching
chfagg T stable 1.5 0.5                   # two heavy tails: split, each half tilted
```

A totally skewed stable law on a tilted grid

The law `stable 1.5 1` at unit scale is the limit law of a Pareto book with tail index 1.5. The DecL statement `chfagg S stable 1.5 1`[2] declares it, and `build` returns it updated. The attribute `info` reports the tilt pair (left, right) in e-folds, θ per unit of loss, the mass left outside the window, and the modeled edge error. The window reaches about 2,000 at log2 16, with a bucket of 1/32.

```
s = build('chfagg S stable 1.5 1')
wprint(s.info)
```

```
aggregate object name    S
value_type               loss
claim count              n/a
frequency distribution   n/a
severity distribution    n/a
loss table               n/a
approximate              exact
bs                       1/32
log2                     16
padding                  n/a
sev_calc                 n/a
dsev_bucket              n/a
normalize                False
x_min                    -10.7812
x_max                    2,037.19
premium                  n/a
expected loss            -0.0265159
loss ratio               n/a
P(X=0)                   0.00617238
validation_eps           0.0001
reinsurance              none
occurrence reinsurance   no reinsurance
aggregate reinsurance    no reinsurance
validation               not unreasonable
family                   stable
parameters               stable 1.5 1
support                  -inf to inf
scale                    1
total mass               0.999996
outside mass             4.34e-06, below 0, above 4.34e-06
tilt                     (0, 8.32)
theta                    (0, 0.00406)
edge error               0.000164
truncation error         0
transfer sigma           n/a
most negative bucket     2.42e-14
buckets per scale        32
checks                   7 ok
frequency tail           none: a characteristic function law
severity tail            none: a characteristic function law
aggregate tail           (-inf, inf), left super-exponential, right power-law
                         [power-law, alpha=1.5, infinite variance]
bounded                  False
id                       F5834EE7
```

[2] From aggregate 1.30.0

In the body the grid agrees with scipy to six figures; the grid's `sf(x)` is the law's survival at $x + h/2$, the bucket's upper edge. At 1,000 scipy already returns zero, the tail constant $Ax^{-1.5}$ is the reference, and the grid matches it to 4×10^{-5}. Beyond the window, which ends near 2,000, the grid reports the mass it leaves out as `outside_above` rather than a number it does not have.

```
u = Stable(1.5, 1)
h = s.bs
for x in [10, 100, 1000]:
    print(f'{x:6d} {s.sf(x):12.5e} {u.sf(x + h / 2):12.5e} {u.tail_constant * (x + h / 2) **
    -1.5:12.5e}')
print(f'{u.sf(5430):.3g}', f'{s.outside_above:.3e}', f'{s.x_max:,.0f}')
```

```
    10  1.25322e-02  1.25322e-02  1.25862e-02
   100  3.98847e-04  3.98847e-04  3.98849e-04
  1000  1.26149e-05  0.00000e+00  1.26154e-05
0 4.339e-06 2,037
```

A totally skewed stable law without tilting

The hint `hints{tilt=False}`, or the call `update(tilt=False)`, inverts the law on the real line and lets it wrap, for comparison with the tilted grid. The window is sized to hold the mass budget on both sides and has the same length as the tilted window, but the modeled error at its right edge is 0.32, compared with 1.6×10^{-4} on the tilted grid. Against the tilted answer, the untilted survival is two parts in a thousand low at 100 and a quarter low at 1,000.

```
w = build('chfagg W stable 1.5 1 hints{tilt=False}')
print(w.tilt, f'{w.edge_error:.2f}', f'{s.edge_error:.1e}')
for x in [100, 1000]:
    print(f'{x:6d} {w.sf(x):12.5e} {s.sf(x):12.5e}')
```

```
(0.0, 0.0) 0.32 1.6e-04
   100  3.97998e-04  3.98847e-04
  1000  9.35487e-06  1.26149e-05
```

A number given as the argument `tilt` is the tilt in e-folds: `tilt=8` damps the wrap by e^{-8} and amplifies roundoff by e^{8}. The automatic choice sits near 8 here, so the two grids are close.

```
w.update(tilt=8)
print(w.tilt, s.tilt, f'{np.abs(w.agg_density - s.agg_density).max():.1e}')
```

```
(0.0, 8.0) (0.0, 8.323062028545595) 9.2e-15
```

Coverage, the bucket cap, and the recommended grid

Because the tilt amplifies the truncation error too, the bucket is capped at $\min(\gamma/10,\ \pi\gamma/(52 \log 2)^{1/\alpha})$, and a small α makes the cap bind: at $\alpha = 0.7$ it is $1/64$, the window is 1,024 scales long at log2 16, and half a percent of the law lies beyond it. The grid is accurate where it reaches, to 10^{-6} in the body, and the `checks` line of `info` flags the coverage. Where a larger grid would cover the budget that line gives it as a paste-ready `hints{}` clause; here none up to log2 24 does, since the budget of 10^{-5} needs a window of 10^{7} scales, and the line reports that. The verdict is `grid infeasible for this law`, with the deficit. The method `sharpen` moves toward the recommended grid one `log2` per call, confirming a grid once its outside mass is within the budget.

```
t = build('chfagg T stable 0.7 1')
wprint(t.validation_description)
print('\n'.join(ln for ln in t.info.splitlines() if ln.startswith(('total mass', 'outside mass',
'edge error', 'tilt', 'checks'))))
print(t.recommended_grid())
```

```
grid infeasible for this law; fails pmf deficit 5.792e-03
total mass              0.994208
outside mass            0.00579, below 0, above 0.00579
tilt                    (0, 12.3)
edge error              9.83e-06
checks                  6 ok, 1 flag: outside mass, coverage needs log2 above 24; log2=24 with
bs=0.015625 is the most this engine offers
(24, 0.015625)
```

At $\alpha = 1/2$ and $\beta = 1$ the stable law is the Lévy law, which scipy has in closed form, so every bucket can be checked. The masses agree to 10^{-6} relative across the body and the outside mass equals the exact survival at the window's end.

```
l = AggregateCHF('L', 'stable', alpha=0.5, beta=1)
l.update()
lv = ss.levy()
exact = lv.cdf(l.xs + l.bs / 2) - lv.cdf(l.xs - l.bs / 2)
body = exact > 1e-6
print(f'{(np.abs(l.agg_density - exact)[body] / exact[body]).max():.1e}',
      f'{l.outside_above:.6f}', f'{lv.sf(l.x_max + l.bs / 2):.6f}')
```

```
2.5e-07 0.070432 0.070432
```

A law with a mean and a power tail has a window mean short of the theoretical mean by the tail beyond the window, at any grid size. The verdict therefore reads the completed mean, the window mean plus that tail in closed form, which recovers the theoretical mean: `2 * stable 1.5 1 + 10` has the verdict `not unreasonable`. The attribute `validation_explanation` gives both means. The method `sharpen` leaves the mean out of its score, since no grid under the bucket cap closes the gap in the window mean.

```
c = build('chfagg S 2 * stable 1.5 1 + 10')
print(c.validation_description)
wprint(next(s for s in c.validation_explanation.split('. ') if 'completed mean' in s))
wprint(c.sharpen().sharpen_description)
```

```
not unreasonable
The law has a power tail, so the window mean 9.94692 omits the tail beyond it,
0.0530761 in closed form; the completed mean 10 against the theoretical 10 is
the check the window cannot make
Sharpen: not run, the grid already scores 0.28 against a target of 0.5 and its
outside mass is within the budget.
```

Stable laws skewed to the left and the first bucket

A law with $\beta = -1$ is the negative of one with $\beta = 1$, and is computed that way: the tilt damps its heavy left tail, the window runs from far below to a few scales above the location, and the mass below the window is added to the first bucket, so `cdf` is right across the window. The first bucket means "the left edge and everything below it".

```
m = AggregateCHF('M', 'stable', alpha=1.5, beta=-1)
m.update()
print(m.tilt, f'{m.x_min:,.0f}', f'{m.x_max:,.2f}', f'{m.outside_below:.2e}')
print(f'{m.agg_density[0]:.2e}', f'{m.cdf(-1000):.5e}', f'{s.sf(1000 - h):.5e}')
```

```
(8.323062028545595, 0.0) -2,037 10.78 4.34e-06
4.34e-06 1.26162e-05 1.26155e-05
```

Stable laws with two heavy tails

A law with $|\beta| < 1$ has a power tail on each side, and no single contour serves it: $\mathrm{P}e^{\theta X}$ is infinite for every $\theta \neq 0$. The S1 law splits exactly instead, $X = X_+ + X_-$ with $X_+ \sim S(\alpha, 1, \gamma_+, \delta_1)$ and $X_- \sim S(\alpha, -1, \gamma_-, 0)$, $\gamma_\pm = \gamma((1 \pm \beta)/2)^{1/\alpha}$. Each half has one heavy tail, is tilted toward its light side on twice the grid, and the two are combined by a linear convolution so that nothing wraps; the window kept is the one leaving equal mass below and above, so its left edge is an output rather than an input. `tilt` reports the pair (τ_-, τ_+) and `outside_below` and `outside_above` the balanced masses. Near $|\beta| = 1$ the light half is narrower than a bucket and a normal law is moved from the wide half to the narrow one to resolve it, which `transfer_sigma` reports.

```
two = AggregateCHF('Two', 'stable', alpha=1.5, beta=0.5)
two.update()
print(two.tilt, f'{two.x_min:,.0f}', f'{two.x_max:,.0f}', f'{two.outside_below:.2e}',
f'{two.outside_above:.2e}')
near = AggregateCHF('Near', 'stable', alpha=1.5, beta=0.99)
near.update()
print(near.tilt, f'{near.transfer_sigma / near.bs:.1f}', f'{near.edge_error:.1e}')
```

```
(5.432747148712065, 6.3485194215615826) -1,330 2,766 2.06e-06 2.06e-06
(3.021108943934432, 7.45121996689686) 2.5 2.2e-04
```

Both tails agree with scipy at 50 scales, to six figures, and both are still there at 500, where scipy returns zero on either side. The left tail of `stable 1.5 0.5` has a constant equal to a third of the right tail's constant, $(1 - \beta)/(1 + \beta)$.

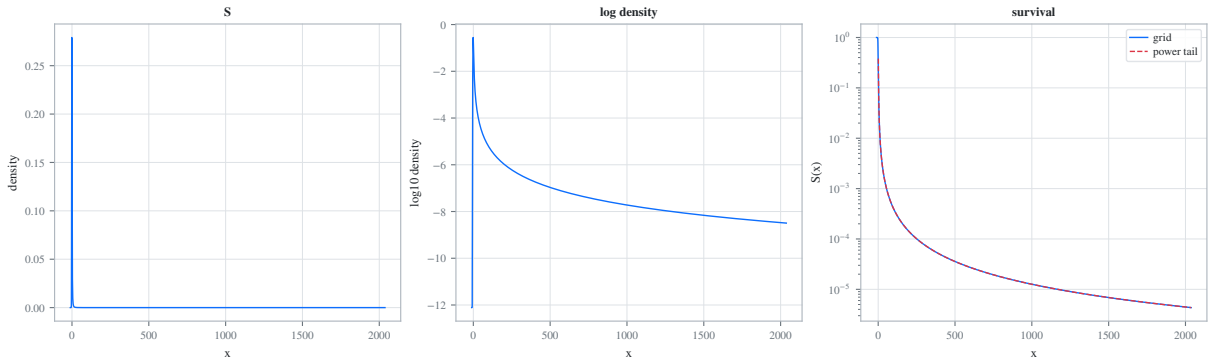
```
v = Stable(1.5, 0.5)
for x in [-500, -50, 50, 500]:
    ref = v.cdf(x + two.bs / 2) if x < 0 else v.sf(x + two.bs / 2)
    got = two.cdf(x) if x < 0 else two.sf(x)
    print(f'{x:6d} {got:12.5e} {ref:12.5e}')
print(f'{v.left_tail_constant / v.tail_constant:.3f}')
```

```
  -500  8.92370e-06  0.00000e+00
   -50  2.84290e-04  2.84289e-04
    50  8.47386e-04  8.47387e-04
   500  2.67608e-05  0.00000e+00
0.333
```

Plotting a stable law on the grid

The method `plot` draws three panels: the density, its logarithm, and the survival function on a log axis with the power tail $Ax^{-\alpha}$ drawn in. The tail and the grid coincide out to the window's end.

```
fig = s.plot()
```



Refused calls

A tilt below zero, or one that is not a number, is refused, as is a numeric tilt on a law with no power tail.

```
for bad in [-1, 'x']:
    try:
        s.update(tilt=bad)
    except ValueError as err:
        wprint(err)
try:
    AggregateCHF('G', 'gamma', shape=4, scale=25).update(tilt=8)
except ValueError as err:
    wprint(err)
```

```
tilt must be True, False or a number >= 0, got -1
tilt must be True, False or a number >= 0, got 'x'
G: tilt=8 on a gamma law, which has no power tail; such a law barely wraps, so
there is nothing for a tilt to damp. Use tilt=True or tilt=False.
```

Using `AllocationBounds`

Ranges of the natural allocation of a portfolio premium to each unit over all distortions that price the total

Stephen J. Mildenhall

2026-11-06

A distortion g that prices the total X of a portfolio to a premium P also allocates P to the units, by the **natural allocation**

$$A_i(g) = \sum_x \kappa_i(x)\big(g(S(x^-)) - g(S(x))\big), \qquad \kappa_i(x) = \mathsf{P}(X_i \mid X = x),$$

where S is the survival function of X, and the allocations add up to P. Different distortions in the set G_P of those pricing $X \wedge a$ to P allocate it differently. The class `AllocationBounds`[1] computes, for each unit, the range of A_i over G_P.

In terms of the Kusuoka measure μ of g, $A_i = \int a_i(p)\,\mu(dp)$, where $a_i(p)$ is the natural allocation of $\mathrm{TVaR}_p(X)$ to unit i. The premium and the allocation are both linear in μ, so the extreme allocations are attained by **biTVaRs**, mixtures of two TVaRs, and the range of A_i is the vertical slice at $\mathrm{TVaR}_X = P$ through the convex hull of the curve $p \mapsto (\mathrm{TVaR}_p(X), a_i(p))$. On the discrete grid the curve is exactly piecewise linear with a vertex at each CDF breakpoint of X, the pair of conditional tail expectations $(\mathsf{P}(X \mid X \ge x_m), \mathsf{P}(X_i \mid X \ge x_m))$, so the bounds are exact. With assets a the default states $X \ge a$ collapse into one outcome at a, which allocates $a\,\mathsf{P}(X_i/X \mid X \ge a)$ to unit i, the equal priority (linear) allocation.

The class `AllocationBounds` is a subclass of the same convex hull engine as `PricingBounds` (Using `PricingBounds`), which bounds the price of a risk on its own rather than its share of the portfolio premium. The class `Bounds` (Using `Bounds`) computes the envelope of the distortions in G_P themselves. Use `AllocationBounds` for the share of each unit. The hull is computed once, and the method `bounds(P)` slices it at any premium in `premium_range`. Because the allocations add up to P, with two units the distortion that gives one unit its largest allocation gives the other its smallest, and the two ranges have the same width.

```
from aggregate.bounds import AllocationBounds
AllocationBounds(port, a=np.inf, units=None, s_floor=1e-14)
port.allocation_bounds(a=0, p=0, units=None)     # p sets a = port.q(p)
```

The portfolio must be updated, so that `density_df` contains the conditional expectations `exeqa_<unit>`. The argument `units` selects units, by default all of them. The method `Portfolio.allocation_bounds` with `a` and `p` both zero leaves the total uncapped.

[1] From aggregate 1.0

```
import numpy as np
from aggregate import build, __version__
from aggregate.bounds import AllocationBounds
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
2.1.0a5
```

Allocation ranges for two units

The example is the portfolio `Two` of Using `Bounds`. Unit `Thin` has 10 claims with a gamma severity of mean 10 and coefficient of variation (CV) 0.5, and unit `Thick` has 2 claims with a lognormal severity of mean 20 and CV 2, limited to 250. Assets are the 99th percentile of the total, $a = 365.125$, and the premium $P = 158.797$ earns a 10% cost of capital. The attribute `info` of `AllocationBounds` shows 5,843 vertices, feasible premiums from $L = \mathsf{P}(X \wedge a) = 138.164$ to a, and an additivity error of 8.4×10^{-9}, the largest gap between TVaR and the sum of its unit allocations over the vertices.

```
port = build('''port Two
  agg Thin  10 claims sev gamma 10 cv 0.5 poisson
  agg Thick  2 claims 250 xs 0 sev lognorm 20 cv 2 poisson''')
a = port.q(0.99)
port.calibrate_distortions(0.10, a=a)
P = port.calibration_df.P.iloc[0]
ab = AllocationBounds(port, a=a)
wprint(ab.info)
GT(ab.bounds(P).droplevel('P'), float_format='.3f')
```

```
allocation bounds object Two
kind                     natural-allocation premium ranges by unit
asset cap                365.125
units                    Thin, Thick
vertices                 5843
premium range            [138.164, 365.125]
s_floor                  1e-14
additivity error         8.37e-09
```

unit	lower	upper	width
Thick	47.518	57.995	10.476
Thin	100.802	111.278	10.476

The allocation to `Thick` lies between 47.518 and 57.995, and to `Thin` between 100.802 and 111.278. Both ranges have width 10.476, and the lower end for `Thin` plus the upper end for `Thick` is P. The width is 22% of the smallest allocation to `Thick` and 10% of the smallest to `Thin`.

Allocations under the five calibrated distortions

Each of the five calibrated distortions is in G_P. The method `port.price(a=a)` computes its natural allocation of P (see Using `price`). The table lists the distortions in order of their Gini index, `gini`. Each allocation lies in the range, to within 0.001 at the ends: `tvar` gives `Thick` its smallest allocation and `ccoc` its largest, and here the allocation to `Thick` falls as `gini` rises.

```
alloc = port.price(a=a).df.P.unstack('unit').loc[port.distortion_df.index]
alloc.insert(0, 'gini', port.distortion_df.gini)
GT(alloc, float_format='.3f', formatters={'gini': '{x:.4f}'})
```

distortion	gini	Thick	Thin	total
ccoc	0.0909	57.995	100.801	158.797
ph	0.1556	53.978	104.819	158.797
wang	0.1868	51.814	106.983	158.797
dual	0.2196	49.511	109.286	158.797
tvar	0.2461	47.518	111.278	158.797

Distortions that attain the allocation bounds

The method `bitvars(P)` lists the biTVaR at each end of each range, as `p0`, `p1`, and the weight `w1` on `p1`. The upper end for `Thick` and the lower end for `Thin` share one biTVaR, the mean mixed with TVaR at 0.99, which equals the maximum a of $X \wedge a$, with weight 0.0909: the `ccoc` distortion. The other two ends share a bracket around $p^* = 0.2461$, equivalent to TVaR at p^*. The method `distortion(P, unit, bound)` returns the achieving `Distortion`. The method `check(P)` recomputes each allocation and the total premium from first principles with it. The largest errors are below 10^{-12}.

```
print(ab.check(P)[['err', 'total_err']].abs().max().to_dict())
GT(ab.bitvars(P).droplevel('P'), float_format='.4f')
```

```
{'err': 1.5631940186722204e-13, 'total_err': 6.252776074688882e-13}
```

unit	*bound*	value	p0	p1	w1
Thick	lower	47.5185	0.2457	0.2462	0.9083
	upper	57.9950	0.0000	0.9900	0.0909
Thin	lower	100.8020	0.0000	0.9900	0.0909
	upper	111.2785	0.2457	0.2462	0.9083

The allocation curves

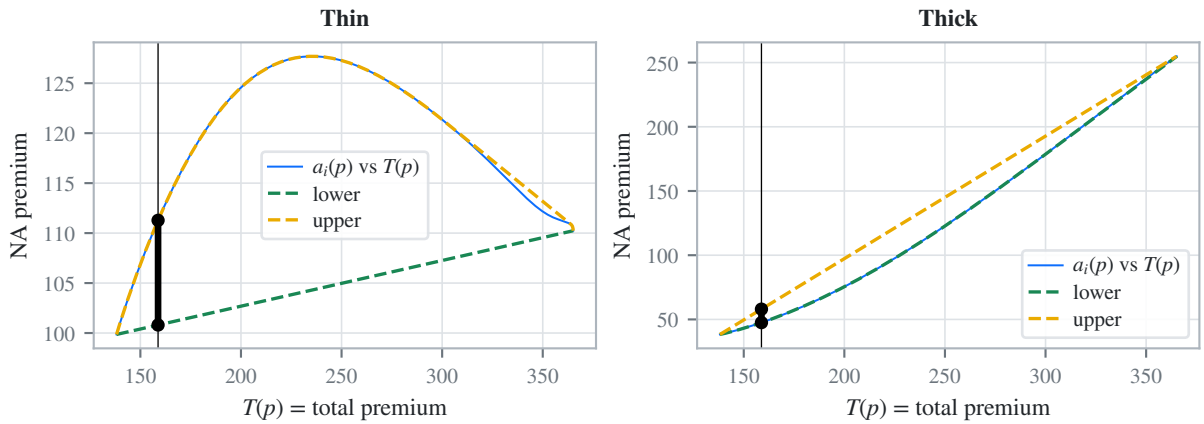
The table `curve_df` lists the vertices, indexed by p, with $\mathrm{TVaR}_p(X \wedge a)$ in `exeqa_total` and $a_i(p)$ in `exeqa_<unit>`. The first row, $p = 0$, contains the expected losses by unit, 99.859 and 38.306, which add up to L. The last is the default outcome at a, which allocates 110.237 to `Thin` and 254.888 to `Thick`.

```
GT(ab.curve_df.iloc[[0, -1]], float_format='.3f')
```

p	exeqa_total	exeqa_Thin	exeqa_Thick
0.000	138.164	99.859	38.306
0.990	365.125	110.237	254.888

The plot shows the curve of each unit, its lower and upper hulls, and the slice at P. The curve for `Thin` rises and then falls: far in the tail the total is large mainly because `Thick` is large, so the allocation of TVaR to `Thin` shrinks. The lower hull of `Thin` and the upper hull of `Thick` are both the chord from the first vertex to the last, whose points are the `ccoc` distortions at different costs of capital.

```
fig, axs = figure('full', cols=2)
ab.plot(P=P, axs=axs);
```



Refused inputs

A unit that is not in the portfolio is refused, and so is a premium outside `premium_range`.

```
refuse(lambda: AllocationBounds(port, a=a, units=['Cat']),
       lambda: ab.bounds(130))
```

```
density_df missing columns ['exeqa_Cat']; check unit names ['Cat'].
P=[130.] outside feasible premium range [138.164, 365.125] (= [E[X], TVaR at
s_floor truncation]).
```

Using `approximate`

Method-of-moments approximations to an aggregate distribution and their error in the tail

Stephen J. Mildenhall

2026-10-30

A **method-of-moments approximation** replaces an aggregate distribution by a continuous distribution from a named family, with parameters chosen to match its mean, coefficient of variation (CV), and, for a shifted family, its skewness. The approximation is quick to state and to evaluate, and it suits a unit with many claims, whose distribution is close to normal. Its weakness is the tail: two distributions with the same three moments can have quite different 1 in 1,000 year losses. The library fits five families, compares each with the FFT answer, and can use any of them in place of the convolution.

With mean m, standard deviation σ, CV $\nu = \sigma/m$, and skewness $\gamma > 0$, the shifted gamma `sgamma` is $s + G$, where G is gamma with shape α and scale θ, and

$$\alpha = \frac{4}{\gamma^2}, \qquad \theta = \frac{\sigma\gamma}{2}, \qquad s = m - \frac{2\sigma}{\gamma}.$$

The shifted lognormal `slognorm` is $s+e^{\mu+\tau Z}$, with Z standard normal and τ solving $\gamma = (\omega+2)\sqrt{\omega-1}$, $\omega = e^{\tau^2}$. The families `gamma` and `lognorm` have no shift and match the mean and CV only, so their skewness is fixed by the CV, at 2ν and $3\nu + \nu^3$. The family `norm` matches the mean and CV with zero skewness. A left-skewed distribution is fitted by reflection.

The DecL clause `approximate KIND`, or `approx KIND`, follows the frequency and any aggregate reinsurance in an `agg` program, where KIND is one of the five families or `exact`, the default. At construction the clause replaces the compound by the fitted distribution, a single severity on a fixed count of one claim, fitted to the theoretical moments of the compound. The attribute `approximation` holds the kind, or `''` for an exact aggregate. On a built `Aggregate` or `Portfolio`, the method `approximate(approx_type, output)` returns a fit to the computed distribution, and the property `approximation_df` compares all five fits with the FFT answer.

```
agg Book 100 claims sev lognorm 100 cv 2 poisson approximate sgamma
agg Book 100 claims sev lognorm 100 cv 2 poisson approx slognorm
agg Book 100 claims sev lognorm 100 cv 2 poisson aggregate net of 5000 xs 12000 approximate sgamma
approximate norm | gamma | lognorm | sgamma | slognorm | exact     # the six kinds
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

A shifted gamma in place of the convolution

The example book has 100 Poisson claims from a lognormal severity with mean 100 and CV $\nu_X = 2$, for an aggregate mean of 10,000. Its variance is $100 \times 100^2 \times (1 + 2^2) = 5{,}000{,}000$, so $\sigma = 2{,}236$, and for a lognormal severity the skewness is $(1 + \nu_X^2)^{3/2}/\sqrt{100} = 5^{3/2}/10 = 1.118$. The shifted gamma therefore has shape $\alpha = 3.2$, scale $\theta = 1{,}250$, and shift $s = 6{,}000$, the parameters shown in the note of the approximated aggregate. In the qd description the frequency is the fixed count of one, and the severity and the aggregate are the same distribution.

```
book = 'agg {} 100 claims sev lognorm 100 cv 2 poisson {}'
s = build(book.format('BookSG', 'approximate sgamma'))
qd(s)
print(repr(s.approximation))
wprint(s.note)
```

```
Aggregate object: BookSG. Frequency distribution fixed. Severity gamma,
[0, inf), exponential right tail. Updated with bucket size 2 and log2 =
16. Validation: not unreasonable.

      Mean     SD      CV  Skew  P01 Median   P99
X
Freq     1      0       0
Sev  10000 2236.1 0.22361 1.118 6634   9592 16928
Agg  10000 2236.1 0.22361 1.118 6634   9592 16928
'sgamma'
agg BookSG
  100 claims
  sev lognorm 100 cv 2
  poisson
  approximate sgamma  approximated by sgamma: gamma(loc=6000, scale=1250,
                      a=3.2), m=10000 cv=0.223607 skew=1.11803
```

Five approximations against the FFT answer

The property `approximation_df` fits all five families to an aggregate computed by FFT and sets each against the `exact` column, the FFT answer. Its rows form four blocks: `meta`, the fitted shape, shift (`loc`) and scale; `stats`, the mean, CV and skewness of each fit, and `ks`, its Kolmogorov distance $\sup_x |F(x) - G(x)|$ from the FFT distribution; `quantiles`, at probabilities from 0.001 to 0.999; and `rel err`, each quantile as a relative error against the FFT quantile. Every family matches the mean and CV. The fits use the moments of the computed distribution, whose skewness, 1.1108, is a little below the theoretical 1.118 because the grid cuts off the far tail of the lognormal severity.

```
a = build(book.format('Book', ''))
df = a.approximation_df
GT(df.loc['stats'])
```

measure	exact	norm	gamma	lognorm	sgamma	slognorm
mean	10000.0175	10000.0192	10000.0175	10000.0175	10000.0175	10000.0175
cv	0.2236	0.2236	0.2236	0.2236	0.2236	0.2236
skew	1.1108	0.0000	0.4472	0.6820	1.1108	1.1108
ks	0.0000	0.0509	0.0234	0.0114	0.0301	0.0206

In the right tail the two-parameter fits are too light, the more so the further out: at 0.999 the normal is 20.8% below the FFT quantile, the gamma 14.1%, and the lognormal 9.6%. Matching the skewness helps most far out: at 0.999 the shifted fits are 4.1% and 2.1% low, while at 0.99 they are 2.1% and 2.2% high, a little further off than the lognormal. The left tail behaves differently, because the shift bounds each shifted fit below, and their 0.01 quantiles are 11.5% and 7.1% too high. The lognormal, which does not match the skewness, has the smallest Kolmogorov distance, 0.011: matching the third moment improves the right tail but does not improve every measure of fit.

```
GT(df.loc['rel err'].loc[[0.01, 0.1, 0.5, 0.9, 0.99, 0.999]], ratio_cols='all')
```

measure	exact	norm	gamma	lognorm	sgamma	slognorm
0.01	0.0%	-19.2%	-6.7%	-1.8%	11.5%	7.1%
0.1	0.0%	-4.1%	-2.4%	-1.2%	1.2%	1.1%
0.5	0.0%	2.6%	0.9%	0.2%	-1.5%	-1.1%
0.9	0.0%	0.2%	0.9%	0.9%	1.3%	0.7%
0.99	0.0%	-8.3%	-3.9%	-1.6%	2.1%	2.2%
0.999	0.0%	-20.8%	-14.1%	-9.6%	-4.1%	-2.1%

Approximation error and the expected claim count

The skewness of a compound Poisson distribution falls as $1/\sqrt{\lambda}$ in the expected claim count λ, so an approximation improves as the count grows. The table repeats the relative errors at 0.99 and 0.999 for the same severity at 10, 100, 1,000, and 10,000 expected claims. At 10,000 claims every family is within 0.4%. At 10 claims the normal is 49% low at 0.999, and the shifted gamma is 15% high at 0.99. With skewness 3.53 its shape is $4/3.53^2 = 0.32$, and a gamma with shape below one has a density that is infinite at the shift, a poor model of a book of ten claims.

```
families = ['norm', 'gamma', 'lognorm', 'sgamma', 'slognorm']
rows = {}
for n in [10, 100, 1000, 10000]:
    d = build(f'agg N{n} {n} claims sev lognorm 100 cv 2 poisson').approximation_df
    for p in [0.99, 0.999]:
        rows[(n, p)] = [d.loc[('stats', 'skew'), 'exact'],
                        *d.loc[('rel err', p), families]]
err = pd.DataFrame.from_dict(rows, orient='index', columns=['skew', *families])
err.index = pd.MultiIndex.from_tuples(err.index, names=['claims', 'p'])
GT(err, ratio_cols=families, formatters={'skew': '.3f'})
```

claims	*p*	skew	norm	gamma	lognorm	sgamma	slognorm
10	0.9900	3.531	-23.8%	-4.4%	3.4%	15.0%	7.4%
	0.9990	3.531	-48.8%	-25.8%	-6.1%	4.2%	5.9%
100	0.9900	1.111	-8.3%	-3.9%	-1.6%	2.1%	2.2%
	0.9990	1.111	-20.8%	-14.1%	-9.6%	-4.1%	-2.1%
1,000	0.9900	0.353	-1.5%	-0.9%	-0.6%	0.0%	0.1%
	0.9990	0.353	-3.8%	-2.7%	-2.0%	-1.0%	-0.8%
10,000	0.9900	0.112	-0.2%	-0.1%	-0.1%	-0.0%	-0.0%
	0.9990	0.112	-0.4%	-0.3%	-0.2%	-0.1%	-0.1%

The approximate method

The method `approximate(approx_type, output)` fits one family to a built aggregate, by default the shifted lognormal, and returns it in the form given by `output`. The default, `'scipy'`, is a frozen `scipy.stats` distribution. The forms `'sev_decl'` and `'agg_decl'` are DecL, a severity clause or a complete program on a fixed count of one claim, and `'agg'` builds that program. With `approx_type='all'` the result is a dictionary with one entry per family. The method fits the moments of the computed distribution, as `approximation_df` does, so the 0.999 quantile of the frozen shifted lognormal equals the `slognorm` entry in the `quantiles` block.

```
fz = a.approximate()
print(f'{fz.ppf(0.999):,.1f}  {df.loc[("quantiles", 0.999), "slognorm"]:,.1f}')
wprint(a.approximate('sgamma', 'agg_decl'))
```

```
20,917.6  20,917.6
agg sgam.Book 1 claim sev 1241.8670417296307 * gamma 3.2417855883847233 +
5974.150788235019 fixed
```

Approximating one unit of a portfolio

A portfolio computes every unit on one grid, and a unit with many small claims can be poorly served by a bucket chosen for the others. In the portfolio below the bucket is 40, which suits the unit `Large`, ten claims with mean 5,000, but not `Small`, 2,000 claims with mean 50. Computed exactly, the severity of `Small` has mean 48.1 on the grid, its aggregate mean is 96,230 rather than 100,000, and validation fails. With `approximate sgamma`, `Small` is one smooth distribution with standard deviation 4,031, which a bucket of 40 resolves, and the portfolio validates.

```
port = '''port {}
  agg Small 2000 claims sev lognorm 50 cv 1.5 poisson {}
  agg Large 10 claims sev lognorm 5000 cv 1 poisson'''
ports = {name: build(port.format(name, clause))
         for name, clause in [('Exact', ''), ('Mixed', 'approximate sgamma')]}
for name, q in ports.items():
    u = q['Small']
    wprint(f'{name}: bucket {q.bs:.0f}; Small severity mean {u.est_sev_m:,.1f}, '
           f'aggregate mean {u.est_m:,.0f}, sd {u.est_sd:,.0f}; '
           f'validation {q.validation_description}')
```

```
Exact: bucket 40; Small severity mean 48.1, aggregate mean 96,230, sd 4,063;
validation fails sev mean, agg mean
Mixed: bucket 40; Small severity mean 100,000.0, aggregate mean 100,000, sd
4,031; validation not unreasonable
```

On a portfolio, the method `approximate` and the property `approximation_df` work on the total, here the sum of a nearly normal unit and a unit of ten claims. The code prints the shifted lognormal fit to the total as DecL, then tabulates the relative errors. The shifted fits are within 1.6% of the FFT quantiles at 0.999, and the shifted lognormal is within 0.4% at all three probabilities shown.

```
mixed = ports['Mixed']
wprint(mixed.approximate('slognorm', 'agg_decl'))
GT(mixed.approximation_df.loc['rel err'].loc[[0.01, 0.99, 0.999]], ratio_cols='all')
```

```
agg slog.Mixed 1 claim sev 78954.16309631686 * lognorm 0.2721911145567305 +
68066.2108631491 fixed
```

measure	exact	norm	gamma	lognorm	sgamma	slognorm
0.01	0.0%	-11.6%	-6.9%	-4.9%	1.5%	0.1%
0.99	0.0%	-6.2%	-3.9%	-2.7%	0.1%	0.2%
0.999	0.0%	-12.6%	-8.7%	-6.3%	-1.5%	-0.4%

Refused programs

An approximated aggregate has no individual claims, so occurrence reinsurance is refused; aggregate reinsurance applies to the fitted distribution and is allowed. A kind outside the six is refused with the list of kinds.

```
lognormal = 'agg R 100 claims sev lognorm 100 cv 2'
refuse(f'{lognormal} occurrence net of 500 xs 500 poisson approximate sgamma',
       f'{lognormal} poisson approximate weibull')
```

```
DecL: approximate is incompatible with occurrence reinsurance (the
method-of-moments fit bypasses the per-occurrence convolution); use aggregate
reinsurance instead.
DecL: approximate 'weibull' is not recognised; use one of exact, norm, lognorm,
gamma, sgamma, slognorm
```

Using `bivariate`

Joint distribution of two aggregates sharing one claim count, coupled by a copula or a discrete bivariate severity

Stephen J. Mildenhall

2026-11-01

A **bivariate aggregate** is the joint distribution of two aggregate losses that share one claim count. Each of the N claims produces a pair (A_i, B_i), and the two totals are $X = A_1 + \cdots + A_N$ and $Y = B_1 + \cdots + B_N$. Loss and allocated expense (ALAE) on the same claims are the standard example, and damage to two lines from the same storms is another. The DecL kind `bivariate`, with the alias `bv`, builds the joint law of (X, Y) as a `BivariateAggregate`, from which the marginals, the law of the sum $X + Y$, and any conditional law follow.

The per-claim pair has a joint probability matrix S on a two-dimensional grid. With $G_N(z) = \mathrm{P}z^N$ the probability generating function of N, applied entry by entry, the joint aggregate density is $\mathrm{FFT2}^{-1}(G_N(\mathrm{FFT2}(S)))$, the ordinary FFT method with two-dimensional transforms. The dependence between X and Y has two sources,

$$\mathrm{cov}(X, Y) = \mathrm{P}N \,\mathrm{cov}(A, B) + \mathrm{var}(N)\,\mathrm{P}A\,\mathrm{P}B,$$

the dependence within a claim and the shared count. For a Poisson count with mean λ the covariance is $\lambda\,\mathrm{P}AB$.

The per-claim matrix S is given in one of two ways. Two component `agg` statements, each with the one-claim frequency `dfreq [1]` (or `dfreq [0 1] [0.7 0.3]` for a component that only 30% of claims reach), set the marginal severities, and a copula clause couples them: `gumbel` or `clayton` with Kendall's tau, `normal` with the correlation, or `fgm` with Spearman's rho. With no copula clause, or `copula independent`, the pair is independent within a claim. Alternatively, `dbvsev` gives S directly, the two-dimensional analog of `dsev`: dense, as two outcome vectors and a matrix whose rows are indexed by the x outcome, or sparse, as a list of `[x y p]` triples. The shared count is a claim count with a frequency, Poisson by default, or a `dfreq`. The same machinery underlies the reinsurance views in Using grossnet, grossceded and netceded and the shared-event model in Using clash.

```
bivariate NAME 10 claims agg A dfreq [1] sev ... agg B dfreq [1] sev ... copula gumbel 0.4 poisson
bv NAME dfreq [1 2] [0.5 0.5] agg A dfreq [1] sev ... agg B dfreq [1] sev ...   # independent
bv NAME 3 claims dbvsev [1 2] [1 2] [[0.4 0.1] [0.1 0.4]] poisson               # dense
bv NAME dfreq [2] dbvsev [[1 1 0.4] [1 2 0.1] [2 1 0.1] [2 2 0.4]]              # sparse
bv NAME dfreq [2] dbvsev [1 2] [1 2]                                             # uniform
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
1.60.1
```

Loss and ALAE coupled by a copula

The example has ten claims a year, Poisson, each with a lognormal loss of mean 100 and coefficient of variation (CV) 1 and a gamma ALAE of mean 25 and CV 0.8. A Gumbel copula with Kendall's tau 0.4 makes a large loss likely to come with a large expense. The program is written with a placeholder for the copula, so that it can be varied below. The function `qd` shows the grid and the realized correlation of X and Y, then one row for each marginal and a `total` row for $X + Y$.

```
program = '''bivariate LossALAE 10 claims
  agg Loss dfreq [1] sev lognorm 100 cv 1
  agg ALAE dfreq [1] sev gamma 25 cv 0.8
  copula {} poisson'''
b = build(program.format('gumbel 0.4'))
qd(b)
```

```
BivariateAggregate(name='LossALAE', units=['Loss', 'ALAE'],
Copula('gumbel', tau=0.4 -> tau=0.4000), shape=(4096, 256), corr=0.8384)

        Mean     SD      CV    Skew  P01  Median  P99
unit
Loss    1000 447.23 0.44723 0.89435  228     940 2316
ALAE  249.91 101.35 0.40553 0.56276   60     240  525
total 1249.9 535.06 0.42808 0.80253  300    1185 2795
```

The table `dependency_df` gives the covariance and correlation per claim (`Sev`) and in aggregate (`Agg`), with the Kendall tau of the copula. The correlation rises from 0.648 per claim to 0.838 in aggregate, because the shared count adds $\lambda \mathsf{P}A \mathsf{P}B = 10 \times 100 \times 25 = 25{,}000$ to the covariance.

```
GT(b.dependency_df, float_format=',.3f')
```

level	cov	corr	tau
Sev	1,301.016	0.648	0.400
Agg	38,001.287	0.838	—

Effect of the dependence on the sum

The same pair is built below under three copulas with the same marginals, and the table shows the correlations and the law of the total. Under `independent` the loss and ALAE of a claim are independent, yet the aggregate correlation is 0.551, close to the $25{,}000/(447.2 \times 101.2) = 0.552$ of the formula; the small gap is discretization. The Clayton copula concentrates dependence among small claims and the Gumbel among large ones, so with the same tau the Gumbel raises the 1 in 100 total, `P99`, and its tail value at risk, `TVaR99`, the most.

```
rows = {}
for cop in ['independent', 'clayton 0.4', 'gumbel 0.4']:
    bc = build(program.format(cop))
    d, t = bc.dependency_df, bc.total
    rows[cop] = [round(d.loc['Sev', 'corr'], 6) + 0, d.loc['Agg', 'corr'],
                 t.mean(), bc.summary_df.loc['total', 'SD'], t.q(0.99), t.tvar(0.99)]
df = pd.DataFrame.from_dict(rows, orient='index', columns=[
    'corr per claim', 'corr aggregate', 'mean', 'SD', 'P99', 'TVaR99'])
df.index.name = 'copula'
GT(df, float_format=',.1f', formatters={'corr per claim': '{x:.3f}',
                                        'corr aggregate': '{x:.3f}'})
```

copula	corr per claim	corr aggregate	mean	SD	P99	TVaR99
independent	0.000	0.551	1,249.9	510.2	2,700	3,023.6
clayton 0.4	0.326	0.696	1,249.9	522.8	2,735	3,063.6
gumbel 0.4	0.648	0.838	1,249.9	535.1	2,795	3,141.2

The column `independent` of `stats_df` gives the moments that $X + Y$ would have if X and Y were independent, as they are with separate claim counts. Its standard deviation is 458.5, against 510.2 with the shared count and independent pairs, and 535.1 with the Gumbel copula as well.

```
s = b.stats_df.loc['agg']
print(s.loc['cv', 'independent'] * s.loc['mean', 'independent'])
```

```
458.53026072441526
```

The marginals and the total of a bivariate aggregate

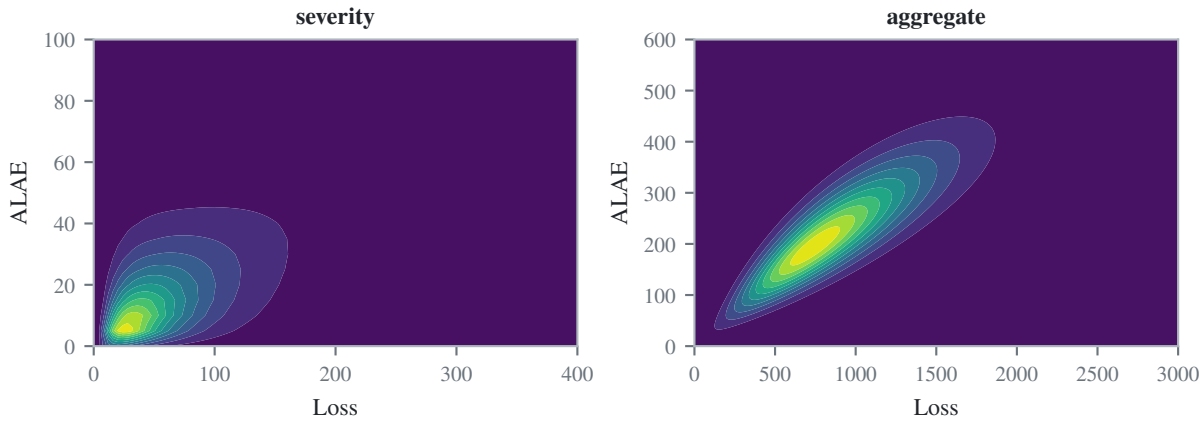
The method `marginal(axis)` returns the law of one axis, selected by number or by name. The attribute `total` returns the law of $X + Y$. Each is a distribution on its grid with `mean`, `q`, `tvar`, and `cdf`. Each marginal reproduces the standalone aggregate of its component. Below, the loss marginal is compared with the ordinary `agg` with the same count and severity: the means agree, and the 99th percentiles, 2,316 and 2,314.5, agree to within the bucket size of the loss axis of the joint grid, 4.

```
m = b.marginal('Loss')
loss = build('agg Loss 10 claims sev lognorm 100 cv 1 poisson')
print(b.bs)
print(m.mean(), loss.est_m)
print(m.q(0.99), loss.q(0.99))
```

```
[4.0, 5.0]
999.9998535035895 999.9999941986528
2316.0 2314.5
```

The method `plot` draws the joint per-claim severity S on the left and the joint aggregate density on the right, as filled contours. The grid extends far into the tail, so the axis limits are set after the call to show the bulk of each law.

```
fig, (ax0, ax1) = figure('full', cols=2)
b.plot(axs=(ax0, ax1))
ax0.set(xlim=[0, 400], ylim=[0, 100])
ax1.set(xlim=[0, 3000], ylim=[0, 600]);
```



A discrete bivariate severity

With `dbvsev` the joint per-claim law is given outright. Below, there are exactly two claims, each a pair on $\{1, 2\}^2$ whose components are equal with probability 0.8, given once as a dense matrix and once as sparse triples. The two forms give the same joint law, and `density_df` shows it with the x outcomes as the rows. By hand, $X = Y = 3$ needs a 1 and a 2 on each axis: the pairs $(1, 1)$ and $(2, 2)$ in either order, $2 \times 0.4^2 = 0.32$, or $(1, 2)$ and $(2, 1)$, $2 \times 0.1^2 = 0.02$, for a total of 0.34.

```
dense = build('bv Dense dfreq [2] dbvsev [1 2] [1 2] [[0.4 0.1] [0.1 0.4]]')
sparse = build('bv Sparse dfreq [2] dbvsev [[1 1 0.4] [1 2 0.1] [2 1 0.1] [2 2 0.4]]')
print(abs(dense.density - sparse.density).max())
GT(dense.density_df.loc[2:4, 2:4], float_format='.4f')
```

```
0.0
```

X	2.0	3.0	4.0
2	0.1600	0.0800	0.0100
3	0.0800	0.3400	0.0800
4	0.0100	0.0800	0.1600

The per-claim correlation is $0.15/0.25 = 0.6$, and with a fixed count the aggregate correlation is the same. A Poisson count with mean 3 adds the shared count term and raises it to $\mathrm{P}AB/\mathrm{P}A^2 = 2.4/2.5 = 0.96$.

```
pois = build('bv Pois 3 claims dbvsev [1 2] [1 2] [[0.4 0.1] [0.1 0.4]] poisson')
print(dense.corr, pois.corr)
```

```
0.5999999999999979 0.9599999992543694
```

Refused programs

A bivariate has exactly two components, a copula parameter must lie in the range of its kind, the copula kind must be one of those listed in the message, and a `dbvsev` matrix may not have negative entries. A pinned grid must hold every `dbvsev` outcome, and the message gives the `log2` that does.

```
import warnings
two = 'agg A dfreq [1] sev gamma 10 cv 1 agg B dfreq [1] sev gamma 10 cv 1'
with warnings.catch_warnings():
    warnings.simplefilter('ignore')    # R5 warns that the pinned grid is short, then refuses
    refuse(f'bv R1 10 claims {two} agg C dfreq [1] sev gamma 10 cv 1 poisson',
           f'bv R2 10 claims {two} copula gumbel 1.4 poisson',
           f'bv R3 10 claims {two} copula student 0.4 poisson',
           'bv R4 dfreq [2] dbvsev [1 2] [1 2] [[0.6 -0.1] [0.1 0.4]]',
           lambda: build('bv R5 dfreq [2] dbvsev [10 20 30] [10 20]', bs=1, log2=(4, 4)))
```

```
bivariate (copula) requires exactly two components; got 3
gumbel copula: tau must be in [0, 1), got 1.4
Unknown copula kind 'student'; available: ['clayton', 'fgm', 'gumbel',
'independent', 'normal', 'shuffle']
dbvsev: probability matrix has negative entries.
axis 0 needs log2 >= 5 at bs 1 to hold the atom 30; log2 4 was given
```

Using Bounds

Pricing bounds: the envelope of all distortions that price a risk to a given premium with given assets

Stephen J. Mildenhall

2026-11-03

A premium P for a risk X supported by assets a does not determine the pricing distortion. Any distortion g with

$$g[X \wedge a] = \int_0^a g(S(x))\,dx = P,$$

where S is the survival function of X, is consistent with it. Square brackets mark g acting on a random variable, as a spectral risk measure, and parentheses mark g acting on a probability. The class `Bounds` computes the set G_P of these distortions and its **envelope**: the smallest and largest value of $g(s)$ at each s over G_P. The method is from Mildenhall (2022), "Similar risks have similar prices: a useful and exact quantification", *Insurance: Mathematics and Economics* 105.

Every concave distortion is a mixture of TVaR distortions, $g(s) = \int \min(1, s/(1-p))\,\mu(dp)$, and then $g[X \wedge a] = \int \mathrm{TVaR}_p(X \wedge a)\,\mu(dp)$. The premium is one linear condition on μ, so the extreme points of G_P are **biTVaRs**,

$$g = (1-w)\,\mathrm{TVaR}_{p_0} + w\,\mathrm{TVaR}_{p_1}, \qquad p_0 \le p^* < p_1,$$

where $\mathrm{TVaR}_{p^*}(X \wedge a) = P$ and the weight w makes g price $X \wedge a$ to P. The class `Bounds` evaluates every bracket (p_0, p_1) from a grid of 256 values of p, refined near p^*, at 513 values of s. A minimum of concave functions is concave, so the lower edge, `min_envelope`, is a `Distortion`. The upper edge, `max_envelope`, is not concave in general and is a plain function of s.

The class `Bounds` is one of three classes in `aggregate.bounds`[1]. It computes the set G_P itself, for one premium fixed when it is built. The class `PricingBounds` computes the exact range of the price of another risk over G_P (Using `PricingBounds`), and `AllocationBounds` the range of the natural allocation of P to each unit of a portfolio (Using `AllocationBounds`). Those two are subclasses of one convex hull engine and take the premium at call time. Use `Bounds` to see which distortions are admissible, and the other two for the price of a particular risk or unit.

```
from aggregate.bounds import Bounds
Bounds(obj, premium, a=np.inf, unit='total', n_p=256, n_s=513)
```

The risk `obj` is a `Portfolio` (its total, or the unit given by `unit`), an `Aggregate`, a `PnL` (its loss engine), or a probability `Series` indexed by outcome. The premium must lie between the mean of X and a.

[1] All three from aggregate 1.0

```
import numpy as np
import pandas as pd
from aggregate import build, __version__
from aggregate.bounds import Bounds, PricingBounds
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
2.0.0
```

A two-unit portfolio priced at a 10% cost of capital

The example here, and on the pages for `PricingBounds` and `AllocationBounds`, is the portfolio `Two`. Unit `Thin` has 10 claims a year with a gamma severity of mean 10 and coefficient of variation (CV) 0.5, and unit `Thick` has 2 claims with a lognormal severity of mean 20 and CV 2, limited to 250. Assets are the 99th percentile of the total, $a = 365.125$. The method `calibrate_distortions` sets the premium that earns a 10% cost of capital on $Q = a - P$, so that $P = L + 0.1(a - P)$ with expected loss $L = \mathsf{P}(X \wedge a)$, and calibrates the five standard distortions to it (see Using distortions).

```
port = build('''port Two
  agg Thin  10 claims sev gamma 10 cv 0.5 poisson
  agg Thick  2 claims 250 xs 0 sev lognorm 20 cv 2 poisson''')
a = port.q(0.99)
port.calibrate_distortions(0.10, a=a)
P = port.calibration_df.P.iloc[0]
GT(port.calibration_df[['L', 'M', 'P', 'Q', 'a']], float_format=',.3f')
```

unit	L	M	P	Q	a
calibration	138.164	20.633	158.797	206.328	365.125

The premium is $P = 158.797$, and the margin $M = 20.633$ is 10% of the capital $Q = 206.328$. A `Bounds` object is built from the portfolio, the premium, and the assets. Its attribute `info` shows $F(a) = 0.99001$ and $p^* = 0.246133$, which is the parameter of the calibrated `tvar` distortion: TVaR at p^* is the one member of G_P that is a single TVaR. The `E[X]` and `margin` rows use the mean of X, 138.667, rather than $L = 138.164$.

```
bd = Bounds(port, P, a=a)
wprint(bd.info)
print(port.distortion_df.loc['tvar', 'param'])
```

```
bounds object name      Two.total
kind                    pricing bounds (IME 2022)
unit                    total
E[X]                    138.667
premium                 158.797
margin                  20.1304
asset cap               365.125
F(a)                    0.99001
p_star                  0.246133
n_p                     256
n_s                     513
0.24613351486398685
```

The bracketing biTVaRs

The table `weight_df` has one row per bracket, indexed by `p_lower` and `p_upper`, with the TVaR of $X \wedge a$ at each end, `t_lower` and `t_upper`, and the `weight` w on p_1. There are 13,593 brackets. The bracket $(0, 1)$ mixes the mean and the maximum, a, with weight $0.0909 = 0.1/1.1$, and it is the calibrated `ccoc` distortion, whose mass at $s = 0$ has the same value. The method `distortion(p0, p1)` returns the biTVaR of a bracket as a `Distortion`, and each prices $X \wedge a$ to P.

```
print(len(bd.weight_df), bd.weight_df.loc[(0.0, 1.0), 'weight'],
      port.distortions['ccoc'].mass)
for p0, p1 in [(0.0, 1.0), (0.125, 0.5)]:
    d = bd.distortion(p0, p1)
    print(d, d.price(port.density_df.p_total, a=a).ask)
```

```
13593 0.09090909090908855 0.09090909090909094
bitvar(0, 1; 0.0909) 158.79693291580026
bitvar(0.125, 0.5; 0.293) 158.796932915786
```

The table `cloud_df` contains $g(s)$ for every bracket, one column per bracket and one row per value of s, and the envelope is its row minimum and maximum. Each calibrated distortion prices $X \wedge a$ to P, so it lies in the band. For s below about 0.17 the lower edge is TVaR at p^*, and above 0.33 it is `ccoc`.

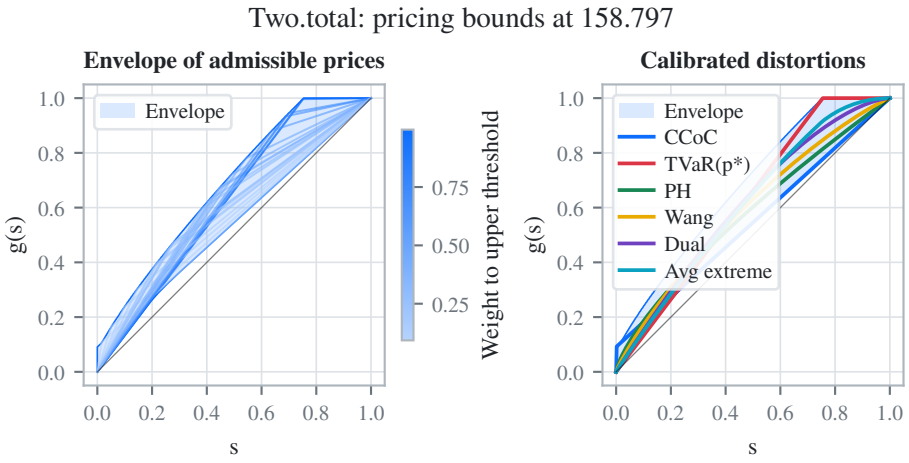
```
s = [1/16, 1/8, 1/4, 1/2, 3/4]
env = pd.DataFrame({'min': bd.cloud_df.min(axis=1).loc[s],
                    'max': bd.cloud_df.max(axis=1).loc[s]})
for k, d in port.distortions.items():
    env[k] = d.g(np.array(s))
GT(env, float_format='.4f')
```

s	min	max	ccoc	ph	wang	dual	tvar
0.0625	0.0829	0.1835	0.1477	0.1319	0.1151	0.0959	0.0829
0.1250	0.1658	0.2753	0.2045	0.2188	0.2072	0.1884	0.1658
0.2500	0.3160	0.4394	0.3182	0.3632	0.3668	0.3621	0.3316
0.5000	0.5455	0.7328	0.5455	0.6026	0.6309	0.6615	0.6632
0.7500	0.7727	0.9962	0.7727	0.8104	0.8434	0.8854	0.9949

Plotting the envelope of distortions

The method `plot_envelope` draws two unit squares. The left contains the band between the edges, with a sample of `n_resamples` bracketing biTVaRs inside it, shaded by weight. The right contains the band again, with the five calibrated distortions and the average of the brackets, `Avg extreme`. With `reflect=True` both panels show the envelope of the dual distortions instead.

```
fig = bd.plot_envelope(n_resamples=20)
```



Bounds on the price of a layer

The envelope bounds the price of any other risk Y over G_P, because $g[Y] = \int g(S_Y(y))\,dy$ and g lies between the edges at every s. For an aggregate layer l xs r on the total, $S_Y(y) = S(r + y)$ for $0 \le y < l$. The bounds hold but need not be sharp, because no one member of G_P attains an edge at every s. The class `PricingBounds` computes the exact range, shown beside the envelope bounds for two layers, 50 xs 150 and 100 xs 250.

```
x, S, bs = port.density_df.loss, port.density_df.S, port.bs
lower = bd.cloud_df.min(axis=1).to_numpy()
layers, rows = {}, {}
for r, l in [(150, 50), (250, 100)]:
    name = f'{l} xs {r}'
    layers[name] = port.density_df.p_total.groupby(np.clip(x - r, 0, l)).sum()
    Sy = S.loc[r:r + l - bs].to_numpy()
    rows[name] = {'envelope lower': np.interp(Sy, bd.s_grid, lower).sum() * bs,
                  'envelope upper': bd.max_envelope(Sy).sum() * bs}
exact = PricingBounds(port, layers, a=a).bounds(P).droplevel('P')
GT(pd.DataFrame(rows).T.rename_axis('layer')
     .join(exact[['lower', 'upper']].add_prefix('exact ')), float_format='.3f')
```

layer	envelope lower	envelope upper	exact lower	exact upper
50 xs 150	13.792	19.770	14.098	18.497
100 xs 250	3.810	12.789	3.810	11.811

On 100 xs 250, S runs from 0.013 to 0.054, where the lower edge is the single distortion TVaR at p^*, so the envelope lower bound, 3.810, is exact. The envelope upper bound, 12.789, is 0.98 above the exact 11.811. On 50 xs 150, S runs from 0.13 to 0.34, across the point where the lower edge changes distortion, and both envelope bounds are loose.

Refused inputs

A premium below the mean of X or above the assets is refused, and so is a unit that is not in the portfolio. The premium test uses the mean of X, 138.667, and not L.

```
refuse(lambda: Bounds(port, 130, a=a),
       lambda: Bounds(port, 400, a=a),
       lambda: Bounds(port, P, a=a, unit='Cat'))
```

```
premium 130.0 below mean 138.66655487886732; pricing bound undefined
premium 400.0 exceeds asset cap 365.125
unit 'Cat' not in portfolio 'Two'
```

Using `clash`

Clash covers on two units hit by the same events with the clash statement

Stephen J. Mildenhall

2026-10-30

A **clash** model describes two units hit by the same events, such as two lines of business exposed to the same storms. An event can reach unit A, unit B, both, or neither, and one that reaches both is a **clash event**. The DecL statement `clash` specifies the model by three expected counts a year, n_a events that reach A only, n_b that reach B only, and n_c that reach both, and a severity for each unit. It returns a `BivariateAggregate`, the joint distribution of the annual losses X and Y of the two units, described in Using `bivariate`.

Within an event the two units are reached independently, with probabilities p_a and p_b. Independence fixes the fourth cell of the two by two table of expected counts, the events that reach neither unit, through $n_c n_0 = n_a n_b$. Hence

$$n_0 = \frac{n_a n_b}{n_c}, \qquad n = n_a + n_b + n_c + n_0, \qquad p_a = \frac{n_a + n_c}{n}, \qquad p_b = \frac{n_b + n_c}{n},$$

where n is the expected shared event count. Each unit becomes an `agg` with the Bernoulli frequency `dfreq [0 1] [1-p p]`, and the two are coupled by the independent copula on the shared count. All the dependence between X and Y comes from shared events. For unit severity means m_A and m_B and a gamma mixed Poisson count with mixing coefficient of variation c,

$$\operatorname{cov}(X, Y) = \left(n_c + c^2 (n_a + n_c)(n_b + n_c)\right) m_A m_B,$$

where the first term is the clash events and the second the common shock of the mixing; under a Poisson count, $c = 0$.

Each unit is given by an optional limit clause and a severity clause, and the shared frequency follows, Poisson by default. A **clash cover** pays per event on the combined loss of both units, in excess of a retention. The `clash` statement computes the annual pair rather than the combined loss of each event, so the cover is built on this page from the per-event components.

```
clash Cat 8 5 2 claims sev lognorm 50 cv 0.6 sev lognorm 60 cv 0.8 mixed gamma 0.2
clash Cat 8 5 2 claims 500 xs 0 sev lognorm 50 cv 0.6 500 xs 0 sev lognorm 60 cv 0.8 poisson
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
1.43.1
```

Shared event count and per-event triggers

The example is the entry `BivariateClash` of the recipe library (see Using recipes): 8 events a year reach A only, 5 reach B only, and 2 reach both, with lognormal severities of means 50 and 60 and coefficients of variation 0.6 and 0.8, and a gamma mixed count with mixing coefficient of variation 0.2. The attribute `clash` holds the solved model. There are $8 \times 5/2 = 20$ events a year that reach neither unit, the shared count has mean 35, and the triggers are $p_a = 10/35 = 0.286$ and $p_b = 7/35 = 0.2$.

```
sevs = 'sev lognorm 50 cv 0.6 sev lognorm 60 cv 0.8 mixed gamma 0.2'
c = build(f'clash Cat 8 5 2 claims {sevs}')
print(c.en)
wprint(c.clash)
```

```
35.0
{'na': 8.0, 'nb': 5.0, 'nc': 2.0, 'n0': 20.0, 'pa': 0.2857142857142857, 'pb':
0.2}
```

The components are named `Cat.A` and `Cat.B`. Unit A is reached by $8 + 2 = 10$ events a year, so its mean annual loss is $10 \times 50 = 500$, and unit B by 7, with mean $7 \times 60 = 420$.

```
qd(c)
```

```
BivariateAggregate(name='Cat', units=['Cat.A', 'Cat.B'],
Copula('independent'), shape=(1024, 1024), corr=0.3120)

       Mean     SD      CV    Skew  P01  Median  P99
unit
Cat.A   500 209.79 0.41959 0.60508  112     480 1076
Cat.B   420    220  0.5238 0.84458   45     390 1060
total   920 348.15 0.37843 0.59124  265     885 1875
```

Dependence from shared events

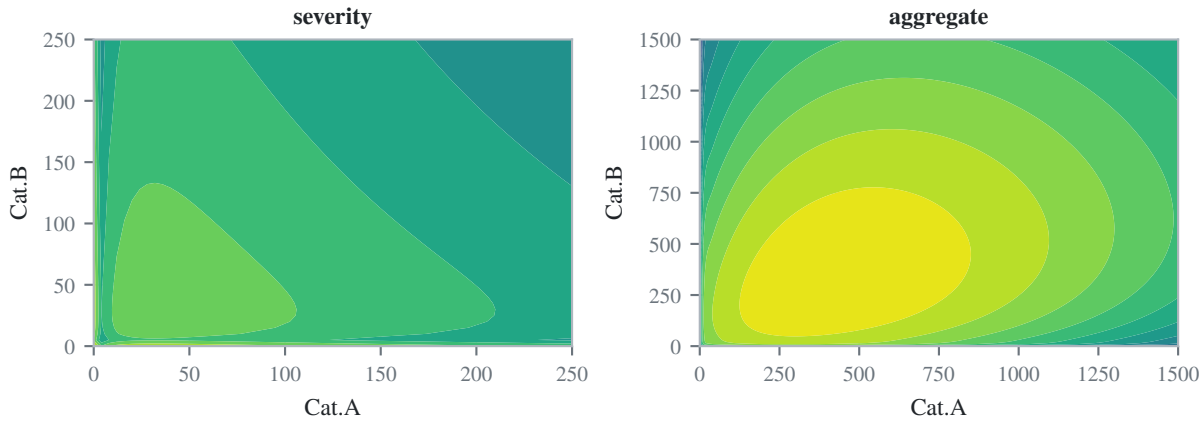
Within an event the two units are independent, so the per-event correlation is zero, while the annual correlation is 0.312. The covariance formula gives $(2 + 0.2^2 \times 10 \times 7) \times 50 \times 60 = 14{,}400$: 6,000 from the two clash events and 8,400 from the mixing.

```
d = c.dependency_df
print(round(d.loc['Sev', 'corr'], 6) + 0, d.loc['Agg', 'corr'])
print(d.loc['Agg', 'cov'], (2 + 0.2**2 * 10 * 7) * 50 * 60)
```

```
0.0 0.31199931266263525
14399.983119090437 14400.000000000002
```

The method `plot` draws the per-event law on the left and the annual joint law on the right, here with a log color scale. On the left, events that reach one unit lie along the axes and clash events fill the interior.

```
fig, (ax0, ax1) = figure('full', cols=2)
c.plot(axs=(ax0, ax1), log=True)
ax0.set(xlim=[0, 250], ylim=[0, 250])
ax1.set(xlim=[0, 1500], ylim=[0, 1500]);
```



A clash cover on the combined event loss

A clash cover `200 xs 150` pays $\min((A+B-150)^+, 200)$ on each event, where A and B are the losses of the two units from that event, either of which may be zero. The per-event combined loss $A+B$ is the total of a portfolio of the two Bernoulli components, with the solved triggers, so an `agg` with the shared count and `sev port.Event` compounds it, and an `occurrence` clause applies the cover. A severity reference needs its source to fix its own grid, hence the `hints` clause. The function below builds the clash, the gross compound and the cover for a set of counts.

```
def clash_cover(counts, layer='200 xs 150'):
    c = build(f'clash Cat {counts} claims {sevs}')
    pa, pb = c.clash['pa'], c.clash['pb']
    build(f'''port Event hints{{log2=12; bs=1/2}}
      agg Event.A dfreq [0 1] [{1 - pa} {pa}] sev lognorm 50 cv 0.6
      agg Event.B dfreq [0 1] [{1 - pb} {pb}] sev lognorm 60 cv 0.8''')
    gross = build(f'agg Gross {c.en} claims sev port.Event mixed gamma 0.2')
    cover = build(f'agg Cover {c.en} claims sev port.Event '
                  f'occurrence ceded to {layer} mixed gamma 0.2')
    return c, gross, cover

results = {counts: clash_cover(counts) for counts in ['8 5 2', '9.5 6.5 0.5']}
```

Without the cover, the compound of the combined event loss is the annual total $X+Y$ of the clash: both have mean 920 and 99th percentile 1,875, to the grid.

```
c, gross, cover = results['8 5 2']
print(c.total.mean(), gross.est_m)
print(c.total.q(0.99), gross.q(0.99))
```

```
919.9995144451415 919.9993396539279
1875.0 1875.5
```

The second set of counts, 9.5, 6.5, and 0.5, keeps each unit's events at 10 and 7, so the two marginals are unchanged, but has a quarter of the clash events. The table shows the annual correlation and the mean, standard deviation, and probability of no recovery of the cover. Fewer clash events lower the correlation from 0.312 to 0.214 and the cost of the cover from 34.99 to 26.04.

```
df = pd.DataFrame.from_dict(
    {k: [c.clash['nc'], c.en, c.corr, cover.est_m, cover.est_sd,
         cover.density_df.p_total.iloc[0]] for k, (c, gross, cover) in results.items()},
    orient='index', columns=['clash events', 'events', 'corr', 'cover mean',
                             'cover SD', 'Pr(no recovery)'])
df.index.name = 'counts'
GT(df, float_format=',.3f')
```

counts	clash events	events	corr	cover mean	cover SD	Pr(no recovery)
8 5 2	2.000	35	0.312	34.993	59.160	0.502
9.5 6.5 0.5	0.500	140	0.214	26.035	51.246	0.598

The same layer bought on each unit separately applies to the loss of one unit in each event, in 10 events a year for A and 7 for B. Together the two pay 23.05 a year, which is what the clash cover would cost with no clash events. Each expected clash event adds $(34.993 - 23.050)/2 = 5.97$ to the cost, and the second set of counts gives the same, $(26.035 - 23.050)/0.5 = 5.97$.

```
per_unit = [build(f'agg {u} {n} claims sev lognorm {m} cv {v} '
                  'occurrence ceded to 200 xs 150 mixed gamma 0.2').est_m
            for u, n, m, v in [('A', 10, 50, 0.6), ('B', 7, 60, 0.8)]]
print(*per_unit, sum(per_unit))
for k, (c, gross, cover) in results.items():
    print(k, (cover.est_m - sum(per_unit)) / c.clash['nc'])
```

```
4.120041844422314 18.929654408650748 23.049696253073062
8 5 2 5.971820469938377
9.5 6.5 0.5 5.971586027564847
```

Refused programs

The clash count must be positive, because $n_0 = n_a n_b / n_c$; two units with no clash events are two separate aggregates. The triggers are independent within an event, so a copula clause is a parse error.

```
refuse(f'clash Cat 10 7 0 claims {sevs}',
       'clash Cat 8 5 2 claims sev lognorm 50 cv 0.6 sev lognorm 60 cv 0.8 copula gumbel 0.4')
```

```
clash: nc (the clash count) must be positive.
DecL parse error at line 1, column 68: Unexpected 'copula'.
```

Using deductibles

Franchise and disappearing deductibles in the DecL layers clause

Stephen J. Mildenhall

2026-10-30

An ordinary deductible `y xs d1` retains the first d_1 of every loss. A franchise deductible retains a loss of at most d_1 and nothing of a larger one. A disappearing deductible sits between them: it retains all of d_1 at a loss of d_1, less as the loss grows, and nothing once the loss reaches the **retention end** d_2. The three are one family of retentions,

$$r(x) = \begin{cases} d_1 & x \le d_1 \\ d_1 \dfrac{d_2 - x}{d_2 - d_1} & d_1 < x \le d_2 \\ 0 & x > d_2, \end{cases}$$

and the layer pays $g(x) = \min((x - r(x))^+, y)$: deductible first, limit second, so y is the largest payment in every form. The ordinary deductible is $d_2 = \infty$, with $g(x) = \min((x - d_1)^+, y)$, and the franchise is $d_2 = d_1$, with $g(x) = \min(x, y)$ for $x > d_1$. In between, g rises with slope $k = d_2/(d_2 - d_1)$ from zero at d_1 to the whole loss at d_2.

The layers clause takes `y xs d1 franchise`[1] and `y xs d1 to d2 disappearing`[2]. The description of the severity shows the form in the layer phrase, as `80 xs 20 franchise`. As for an ordinary layer, the claim count is the count of losses above d_1 and the severity is conditional on $X > d_1$; a `!` after the severity makes it ground up.

```
agg Ordinary 10 claims 80 xs 20 sev 100 * uniform poisson
80 xs 20 franchise                          # in place of 80 xs 20
80 xs 20 to 60 disappearing                 # no retention above 60
[40 80] xs [10 20] to [30 60] disappearing  # one end per layer
```

```
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.7.0
[2] From aggregate 1.11.0

Three deductible forms on a uniform loss

Ten claims a year above 20, from a ground-up loss uniform on $[0, 100]$, under an 80 xs 20 layer, make each deductible form easy to follow. The three aggregates differ only in the layers clause, and the severity in the `qd` description shows the form, here `80 xs 20 franchise`.

```
forms = {'Ordinary': '', 'Franchise': 'franchise',
         'Disappearing': 'to 60 disappearing'}
book = 'agg {} 10 claims 80 xs 20 {} sev 100 * uniform {} poisson'
books = {k: build(book.format(k, v, '')) for k, v in forms.items()}
qd(books['Franchise'])
```

```
Aggregate object: Franchise. Frequency distribution poisson. Severity
uniform, [0, 80], bounded; 80 xs 20 franchise. Updated with bucket size
1/16 and log2 = 16. Validation: not unreasonable.

      Mean     SD      CV     Skew    P01 Median    P99
X
Freq    10 3.1623 0.31623  0.31623
Sev   57.5 19.843  0.3451 -0.32397 20.812     60     80
Agg    575 192.35 0.33453  0.35899 182.44 563.44 1070.4
```

The same layer phrase appears as the declared support in the `tail_explanation` of each severity. The ordinary layer shows `80 xs 20`, with no suffix.

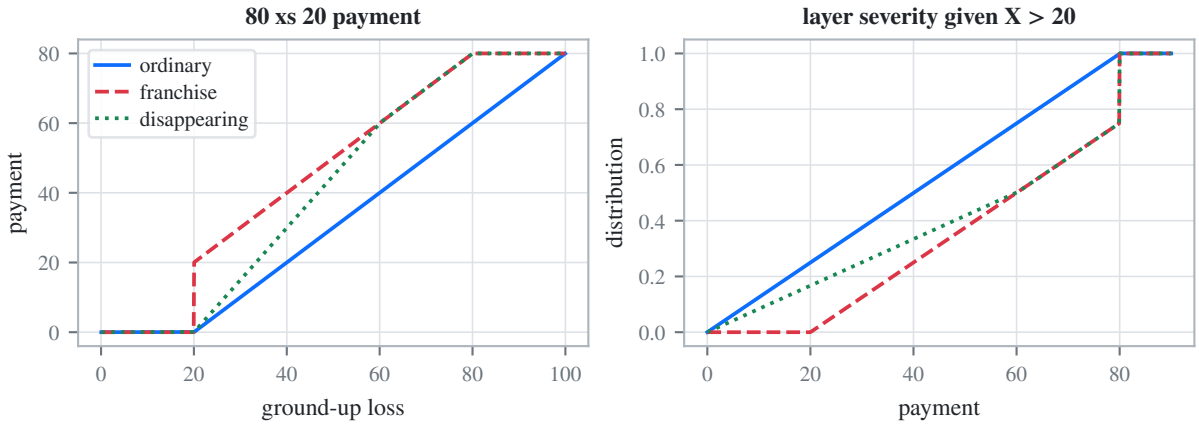
```
for k, a in books.items():
    wprint(a.sevs[0].tail_explanation)
```

```
Uniform severity [0, 80], bounded. Declared support: 80 xs 20. Support is
bounded, so every moment is finite.
Uniform severity [0, 80], bounded. Declared support: 80 xs 20 franchise. Support
is bounded, so every moment is finite.
Uniform severity [0, 80], bounded. Declared support: 80 xs 20 to 60
disappearing. Support is bounded, so every moment is finite.
```

Payment function and layer severity

For a loss uniform on $[0, 100]$ the ground-up quantile at p is $100p$, so the quantile function of the ground-up payment (declared with `!`) is the payment function itself: $g(x)$ is `ppf(x / 100)`. The left panel shows g for each form, and the right panel shows the distribution of the payment given $X > 20$, which is the severity the aggregate convolves.

```
ground_up = {k: build(book.format(k + 'GU', v, '!'), update=False)
             for k, v in forms.items()}
x = np.linspace(0.05, 99.95, 1000)
t = np.linspace(0, 90, 1000)
fig, (ax0, ax1) = figure('full', cols=2)
for (k, a), ls in zip(books.items(), ['-', '--', ':']):
    ax0.plot(x, ground_up[k].sevs[0].ppf(x / 100), ls=ls, label=k.lower())
    ax1.plot(t, a.sevs[0].cdf(t), ls=ls, label=k.lower())
ax0.set(xlabel='ground-up loss', ylabel='payment', title='80 xs 20 payment')
ax1.set(xlabel='payment', ylabel='distribution',
        title='layer severity given X > 20')
ax0.legend();
```



With a franchise deductible the layer pays nothing up to 20 and then the whole loss, so no payment lies in $(0, 20)$. With the disappearing deductible it pays $1.5(x - 20)$ up to 60 and then the whole loss, the same as the franchise above 60. Both reach the limit at a loss of 80, the layer's `detachment`, and put mass 1/4 on it; the ordinary layer reaches the limit only at 100.

```
for k, a in books.items():
    s = a.sevs[0]
    print(f'{k:13s} detachment {s.detachment:4.0f}   '
          f'mass at the limit {s.sf(79.999):.4f}')
```

```
Ordinary      detachment  100   mass at the limit 0.0000
Franchise     detachment   80   mass at the limit 0.2500
Disappearing  detachment   80   mass at the limit 0.2500
```

Layer means against a hand calculation

The hand calculation of the layer means starts from the loss uniform on $[0, 100]$, which given $X > 20$ is uniform on $(20, 100)$. The ordinary layer pays $X - 20$, uniform on $(0, 80)$, mean 40. The franchise layer pays X up to 80 and 80 above, mean $\int_{20}^{80} x\,dx/80 + 80/4 = 57.5$. The disappearing deductible layer has mean $\int_{20}^{60} 1.5(x - 20)\,dx/80 + \int_{60}^{80} x\,dx/80 + 20 = 15 + 17.5 + 20 = 52.5$. The ground-up means are $\Pr(X > 20) = 0.8$ times the layer means: 32, 46, and 42.

```
hand = {'Ordinary': 40, 'Franchise': 57.5, 'Disappearing': 52.5}
df = pd.DataFrame.from_dict(
    {k: [hand[k], a.sevs[0].moms()[0], a.est_sev_m,
         ground_up[k].sevs[0].moms()[0], a.est_m] for k, a in books.items()},
    orient='index',
    columns=['hand', 'layer mean', 'grid mean', 'ground up', 'aggregate'])
df.index.name = 'deductible'
GT(df)
```

deductible	hand	layer mean	grid mean	ground up	aggregate
Ordinary	40.0000	40.0000	40.0000	32.0000	400.0000
Franchise	57.5000	57.5000	57.5000	46.0000	575.0000
Disappearing	52.5000	52.5000	52.5000	42.0000	525.0000

The layer mean is exact, from the same partial expectations as an ordinary layer; the grid mean is that of the discretized severity, on a bucket of 1/16. The aggregate mean is ten times the layer mean.

A vector of layers with disappearing deductibles

In a vector of layers, `to [...]` takes one retention end per layer, and a scalar `to d2` applies to every layer. The first layer below, `40 xs 10 to 30 disappearing`, pays $1.5(x-10)$ on $(10, 30]$ and then $\min(x, 40)$, with mean 33.89 given $X > 10$.

```
v = build('agg Vector 10 claims [40 80] xs [10 20] to [30 60] disappearing '
          'sev 100 * uniform poisson')
for s in v.sevs:
    print(s.tail_description, f'mean {s.moms()[0]:.4f}')
```

```
uniform, [0, 40], bounded; 40 xs 10 to 30 disappearing mean 33.8889
uniform, [0, 80], bounded; 80 xs 20 to 60 disappearing mean 52.5000
```

Refused programs

For a disappearing deductible the retention end must be at least the attachment, and a vector `to` must have one entry per layer. A franchise takes no `to`, since its retention end is the attachment, so `to 60 franchise` is a parse error.

```
refuse(*[p + ' sev 100 * uniform poisson' for p in [
    'agg D 10 claims 80 xs 20 to 10 disappearing',
    'agg D 10 claims [40 80] xs [10 20] to [30 60 90] disappearing',
    'agg D 10 claims 80 xs 20 to 60 franchise']])
```

```
Retention end 10 must be at least the attachment 20.
disappearing deductible: to [...] has 3 entries but there are 2 layers; give one
retention end per layer, or a single value for all of them.
DecL parse error at line 1, column 32: Unexpected 'franchise'.
```

Using distortions

Distortion functions for risk-adjusted pricing with the DecL distortion statement and the Distortion class

Stephen J. Mildenhall

2026-11-06

A **distortion** is an increasing function $g : [0,1] \to [0,1]$ with $g(0) = 0$ and $g(1) = 1$. It prices a loss $X \geq 0$ with survival function $S(x) = \Pr(X > x)$ as

$$g[X] = \int_0^\infty g(S(x))\,dx,$$

which is the expected loss $\mathsf{P}X = \int_0^\infty S(x)\,dx$ when $g(s) = s$. Square brackets mark g acting on a random variable, as a spectral risk measure, and parentheses mark g acting on a probability. A pricing distortion is concave, so $g(s) \geq s$: it moves probability toward the large losses and $g[X] \geq \mathsf{P}X$. Its **dual** $\check{g}(s) = 1 - g(1-s)$ is convex, and $\check{g}[X] \leq \mathsf{P}X$ is the bid price. Every concave distortion is a mixture of TVaR distortions, so $g[X]$ is a weighted average of the TVaRs of X at different levels.

The five families of the standard pricing set each have one shape parameter.

kind	$g(s)$	**parameter**	gini
`ccoc`	$d + (1-d)s$ for $s > 0$	return r, with $d = r/(1+r)$	d
`ph`	s^a	$0 < a \leq 1$	$(1-a)/(1+a)$
`wang`	$\Phi(\Phi^{-1}(s) + \lambda)$	$\lambda \geq 0$	$2\Phi(\lambda/\sqrt{2}) - 1$
`dual`	$1 - (1-s)^b$	$b \geq 1$	$(b-1)/(b+1)$
`tvar`	$\min(s/(1-p), 1)$	$0 \leq p \leq 1$	p

The constant cost of capital `ccoc` has a jump of d at zero, and with assets a it prices $X \wedge a$ at $(1-d)\,\mathsf{P}(X \wedge a) + d\,a$, the premium that earns the return r on the capital. Other important distortions include: `tsallis`, the Beta$(\alpha, 1)$ mixture of TVaRs, $g(s) = (\alpha s - s^\alpha)/(\alpha - 1)$, introduced in Using `tsallis`; `bitvar`, a mixture of TVaRs at $p_0 < p_1$ with weight w_1 on p_1; `wtdtvar`, a weighted TVaR at several levels, built in Python; `beta`; and `cll`, `clin`, `lep` and `ly`, which have a second parameter, the intercept `r0`. The combinators `minimum` and `mixture` take other distortions as their arguments.

The index $\text{gini} = 2\int_0^1 g(s)\,ds - 1$ is twice the area between g and the diagonal, also equal to twice the margin on a standard uniform. A TVaR at level p has $\text{gini} = p$, so the index is the level of the TVaR with the same area. It puts families with incomparable parameters on one scale that increases from 0 for risk neutral to 1 for max pricing. All the tables of distortions on these pages lead with it. The attribute `gini` holds the closed form where a family has one and otherwise integrates g numerically, so every distortion has an index.

```
distortion NAME [as "label"] kind n1 [n2 ...]
distortion Gentle ph 0.6                          # g(s) = s^0.6
distortion Named as "my distortion" ph 0.6        # the same, with a declared label
distortion Cost ccoc 0.1                          # a 10% return on capital
distortion Two bitvar 0.5 0.99 0.25               # p0, p1, then the weight on p1
distortion Low minimum dist.Gentle dist.Two       # the pointwise minimum
distortion Mix mixture dist.Gentle dist.Two wts [0.6 0.4]
```

The numbers after the kind are its parameters in a fixed order, and for `ccoc` the number is the return r. A reference `dist.NAME` copies a stored distortion into a combinator. An `as` clause after the name declares the label, and the attribute `name` stays the family. Pricing with a distortion is the subject of Using `price`, and finding the distortion that a premium implies is the subject of Using `price` and `evaluate`.

```
import numpy as np
import pandas as pd
from aggregate import build, Distortion, __version__
from aggregate.spectral import consistent_distortions
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
2.1.0a5
```

Distortions from DecL and from Python

The DecL statement `distortion` returns a `Distortion`, and the factory `Distortion(kind, shape)` builds the same object in Python. The shape can also be passed by its natural name: `a` for `ph`, `lam` for `wang`, `b` for `dual` and `p` for `tvar`. The static methods `Distortion.ph`, `Distortion.wang` and so on are shortcuts. All four forms below give the same distortion, with the same label `PH(0.6)`, the same index 0.25 and the same `id`.

```
gentle = build('distortion Gentle ph 0.6')
for d in [gentle, Distortion('ph', 0.6), Distortion('ph', a=0.6), Distortion.ph(0.6)]:
    print(f'{type(d).__name__:14s} {d.label:8s} {d.gini:.4f} {d.id()}')
print(gentle.program)
```

```
PHDistortion   PH(0.6)  0.2500 E83B5ED9
PHDistortion   PH(0.6)  0.2500 E83B5ED9
PHDistortion   PH(0.6)  0.2500 E83B5ED9
PHDistortion   PH(0.6)  0.2500 E83B5ED9
distortion Gentle ph 0.6
```

An `as` clause changes the label and leaves the family in `name`:

```
named = build('distortion Named as "my distortion" ph 0.6')
print(named.name, '|', named.label, '|', named.program)
```

```
ph | my distortion | distortion Named as "my distortion" ph 0.6
```

The shortcut for `ccoc` takes the return `r=`, as DecL does, or the discount `d=`, with $d = r/(1+r)$, always by keyword. All four forms below give $d = 1/11$, a 10% return.

```
print(build('distortion Cost ccoc 0.1').d, Distortion('ccoc', r=0.1).d,
      Distortion.ccoc(r=0.1).d, Distortion.ccoc(d=1 / 11).d)
```

```
0.09090909090909091 0.09090909090909091 0.09090909090909091 0.09090909090909091
```

The attribute `info` summarizes a distortion. Its `weights mean` and `weights max` rows are the weights its TVaR mixture puts on the mean ($p = 0$) and on the maximum ($p = 1$). A PH distortion puts weight a on the mean, here 0.6, and none on the maximum; `area` is $\int_0^1 g = (1 + \text{gini})/2$.

```
wprint(gentle.info)
```

```
distortion object name   ph
kind                     ph
kind name                proportional hazard
shape                    0.6
shape name               a
other params             none
weights mean             0.6
weights max              0
interior atoms           False
gini                     0.25
area                     0.625
id                       E83B5ED9
```

Distortions in the recipe library

The recipe library included with aggregate (see Using recipes) holds one distortion of each standard family, each with gini $= 0.2$, and a `minimum` and a `mixture` of two of them. A bare name builds the stored entry. The column `2∫g − 1` is the index computed from g on a fine grid, and it agrees with `gini` for all seven. The index is linear in g, so the mixture of two distortions with index 0.2 has index 0.2, and the minimum lies below both and has a smaller index, 0.162.

```
s = np.linspace(0, 1, 100_001)
lib = {n: build(n) for n in build.discover(kind='distortion', tags='topic:spectral').index}
GT(pd.DataFrame({'gini': [d.gini for d in lib.values()],
                 '2∫g − 1': [2 * np.trapezoid(d.g(s), s) - 1 for d in lib.values()],
                 'label': [d.label for d in lib.values()]},
                index=pd.Index(list(lib), name='recipe')))
```

recipe	gini	2∫g − 1	label
CCoCDistortion	0.2000	0.2000	ccoc(0.25)
DualDistortion	0.2000	0.2000	dual(1.5)
MinimumDistortion	0.1623	0.1623	minimum(2)
MixtureDistortion	0.2000	0.2000	mixture(2)
PHDistortion	0.2000	0.2000	PH(0.667)
TVaRDistortion	0.2000	0.2000	TVaR(0.2)
WangDistortion	0.2000	0.2000	Wang(0.358)

Five families with one Gini index

The function `consistent_distortions(p)` in `aggregate.spectral` returns one distortion of each standard family, all with gini $= p$. They enclose the same area and differ in where they put it. The table shows $g(s)$ at five values of s. At $s = 0.001$, an event in the extreme tail, `ccoc` gives weight 0.251 and PH 0.016, while dual and TVaR give less than 0.002. At $s = 0.5$ `ccoc` gives the least weight, 0.625, and dual the most, 0.685. For the same index, the tail-heavy families `ccoc` and PH load a rare large loss more heavily, and dual and TVaR load the body of the distribution.

```
fams = consistent_distortions(0.25)
ss = [0.001, 0.01, 0.1, 0.5, 0.9]
GT(pd.DataFrame([[d.gini, d.label, *d.g(np.array(ss))] for d in fams.values()],
                columns=['gini', 'label', *[f'g({x})' for x in ss]],
                index=pd.Index(list(fams), name='kind')))
```

kind	gini	label	g(0.001)	g(0.01)	g(0.1)	g(0.5)	g(0.9)
ccoc	0.2500	ccoc(0.333)	0.2507	0.2575	0.3250	0.6250	0.9250
ph	0.2500	PH(0.6)	0.0158	0.0631	0.2512	0.6598	0.9387
wang	0.2500	Wang(0.451)	0.0042	0.0303	0.2030	0.6739	0.9584
dual	0.2500	dual(1.67)	0.0017	0.0166	0.1610	0.6850	0.9785
tvar	0.2500	TVaR(0.25)	0.0013	0.0133	0.1333	0.6667	1.0000

The constructor `Distortion.from_gini(kind, gini)` builds the member of a one-parameter family with a given index, for `ccoc`, `ph`, `wang`, `tsallis`, `dual` and `tvar`, and `consistent_distortions` uses the same inverses. Each distortion below has index 0.25:

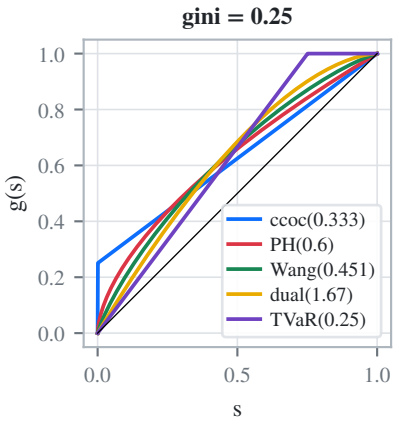
```
for kind in ['ph', 'wang', 'tsallis', 'dual']:
    d = Distortion.from_gini(kind, 0.25)
    print(f'{d.label:12s} {d.gini:.6f}')
```

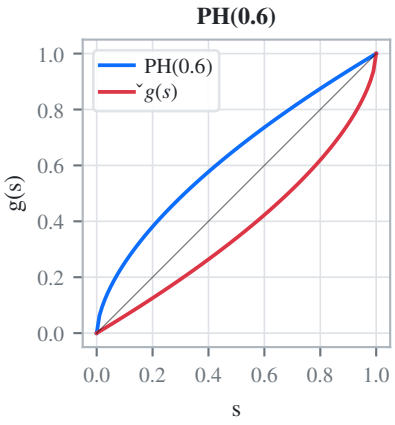
```
PH(0.6)      0.250000
Wang(0.451)  0.250000
Tsallis(3)   0.250000
dual(1.67)   0.250000
```

Plotting a distortion

The left panel shows the five distortions with gini $= 0.25$. The `ccoc` line starts at $d = 0.25$, the jump at zero, and TVaR reaches 1 at $s = 1 - p = 0.75$. The right panel is `Distortion.plot`, which draws g and, by default, its dual $\check{g}$, on axes of equal length, since the concavity of g is what the plot is for.

```
fig, (ax0, ax1) = figure('full', cols=2)
for d in fams.values():
    ax0.plot(s, d.g(s), label=d.label)
ax0.plot([0, 1], [0, 1], c='k', lw=0.5)
ax0.set(xlabel='s', ylabel='g(s)', aspect='equal', title='gini = 0.25')
ax0.legend(fontsize='small')
gentle.plot(ax=ax1);
```





Refused programs

Four distortion programs are refused: `wtdtvar`, which has no DecL form because its parameters are vectors; a `ph` with two parameters; a `bitvar` with $p_0 > p_1$; and a kind that does not exist. The last message lists the available kinds. In Python, `Distortion.ccoc` refuses a bare number, which could be either the return or the discount, and `Distortion.from_gini` refuses a family with more than one parameter, listing its parameters.

```
refuse('distortion W wtdtvar 0.5 0.9',
       'distortion P ph 0.6 0.7',
       'distortion B bitvar 0.9 0.5 0.5',
       'distortion X expo 0.5',
       lambda: Distortion.ccoc(0.1),
       lambda: Distortion.from_gini('bitvar', 0.25))
```

```
Distortion 'wtdtvar' has no DecL number-list form; construct it in Python
(weighted-TVaR takes parameter vectors; the minimum/mixture combinators take
distortion references).
DecL distortion 'ph' expects 1 parameter(s) ('a',), got 2: [0.6, 0.7]
bitvar requires p0 < p1, got p0=0.9, p1=0.5
Unknown distortion kind 'expo'; available: ['beta', 'bitvar', 'ccoc', 'clin',
'cll', 'dual', 'lep', 'ly', 'minimum', 'mixture', 'ph', 'power', 'tsallis',
'tvar', 'wang', 'wtdtvar']
Distortion.ccoc() takes 0 positional arguments but 1 was given
from_gini builds a one-parameter family; 'bitvar' takes parameters ('p0', 'p1',
'w1'). The one-parameter families are ccoc, ph, wang, tsallis, dual, tvar.
```

Using `elt`

Aggregates from event loss tables with the elt clause

Stephen J. Mildenhall

2026-11-07

An event loss table (ELT), the usual output of a catastrophe model, lists events e, each with an annual rate λ_e and a loss X_e given that the event occurs. Each row contains the mean loss μ_e and optionally a maximum M_e, an independent standard deviation (SD) σ_I, and a correlated SD σ_C. **Secondary uncertainty** is the variability of X_e about μ_e, and the two SDs measure it. Events arrive as independent Poisson processes, so the annual loss is compound Poisson with expected count $\Lambda = \sum_e \lambda_e$ and severity the mixture of the X_e with weights λ_e/Λ. Its cumulants are

$$\kappa_r = \sum_e \lambda_e \mathsf{P}X_e^r,$$

and without secondary uncertainty $X_e = \mu_e$ is an atom.

The `elt` clause takes the place of the exposure, severity and frequency clauses of an `agg`, and occurrence and aggregate reinsurance follow as usual. A row is `[id rate mean]` (ELT3), `[id rate mean max]` (ELT4), or `[id rate mean max sdi sdc]` (ELT6). An identifier is an unsigned integer or a quoted string, compared exactly, so `7` and `"7"` are one event and `"007"` another. Each `elt` clause is one source, and `unit NAME [as LABEL]` assigns it to a unit (`Total` when omitted). Rows sharing an identifier across units are one occurrence.

The `elt` clause[1] builds an aggregate from an inline table, as in the programs below. Tables read from CSV and Parquet files are in Using elt with files, and year loss tables (YELT), which list simulated annual losses rather than events, are in Using yelt.

```
agg Basic elt [[1 .1 100] [2 .1 40] [3 .1 80]]
agg Shared elt unit Personal as "Personal lines" [[1 .1 300] [2 .1 100]]
           elt unit Commercial [[1 .1 200] [3 .1 200]]
           occurrence net of 300 xs 200
agg Cat elt no2u [[1 .1 300 3000 400 300] [2 .1 100 1000 371 267]]
```

```
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
2.1.0a6
```

[1] From aggregate 1.12.0

An event loss table of three events

The aggregate `Basic` has three events, each with rate 0.1, so $\Lambda = 0.3$. The mean annual loss is $0.1(100 + 40 + 80) = 22$ and the variance is $0.1(100^2 + 40^2 + 80^2) = 1{,}800$, an SD of 42.43. The severity is three equally weighted atoms, at 40, 80, and 100.

```
a = build('agg Basic elt [[1 .1 100] [2 .1 40] [3 .1 80]]')
qd(a)
```

```
Aggregate object: Basic. Frequency distribution poisson. Severity
dhistogram, [40, 100], bounded; atoms [40 80 100]. Updated with bucket
size 20 and log2 = 5. Validation: not unreasonable.

       Mean      SD      CV    Skew P01 Median P99
X
Freq    0.3 0.54772  1.8257  1.8257
Sev  73.333  24.944 0.34015 -0.3818  40     80 100
Agg      22  42.426  1.9285  2.0637   0      0 180
```

Reports on a loss table aggregate

An aggregate built from a loss table has three reports: `elt_description`, `elt_explanation`, and `elt_df`. The first, `elt_description`, is a few aligned lines: the table, the event rate and mean annual loss, the grid, and the checks counted by status.

```
wprint(a.elt_description)
```

```
loss table              ELT3, 3 events from 3 rows, 1 unit
event rate              0.3, mean annual loss 22
grid                    bs 20, log2 5
checks                  13 pass
```

The frame `elt_df` is the table the model uses, one row per event with a positive rate. The `units` column counts the input rows behind each event.

```
GT(a.elt_df, float_format=',.4g')
```

event	rate	mean	units
1	0.1	100	1
2	0.1	40	1
3	0.1	80	1

The string `elt_explanation` is three paragraphs: the model in words, each source with its provenance, and every check with its value, tolerance, and status, warnings first. The checks compare the moments of the table, the model, and the discretized distribution, and all of them pass here.

```
wprint(a.elt_explanation)
```

```
The annual loss is compound Poisson with event rate 0.3 and severity the
rate-weighted mixture of the event totals, so the mean annual loss is 22. An
event's unit losses are summed into one event total before any occurrence
treaty, and equal totals are combined: the 3 events with a positive rate give 3
severity atoms.

Source 1, unit Total: inline, ELT3, 3 rows.

13 checks: 13 pass. Form projection: pass, 0; 0 ELT6 source(s) read as ELT4
under no2u; combined form ELT3. One row per event and unit: pass, 0 against 0;
no event repeats within a unit. Event rate agreement: pass, 0 against 1e-12;
unit rows whose rate differs from the first row of the event, within the
relative tolerance. Zero-rate events: pass, 0; excluded from the count and the
moments. Severity mass: pass, 0 against 1e-12; lowered severity weight against
the event rate. Severity mean, table against model: pass, 0 against 0.0001;
independent sums before and after coalescing. Annual mean, table against model:
pass, 2.22e-16 against 0.0001; independent sums before and after coalescing.
Severity mean estimate: pass, 0 against 0.0001; discretized density against the
table. Severity cv estimate: pass, 0 against 0.0001; discretized density against
the table. Severity skew estimate: pass, -3.93e-14 against 0.0001; discretized
density against the table. Annual mean estimate: pass, -1.07e-09 against 0.0001;
discretized density against the table. Annual cv estimate: pass, -7.31e-09
against 0.0001; discretized density against the table. Annual skew estimate:
pass, -1.59e-07 against 0.0001; discretized density against the table.
```

Units and shared events

The aggregate `Shared` has two units, Personal (labeled "Personal lines") and Commercial, and event 1 has a row in each. Its losses add to one occurrence of 500 before the occurrence treaty 300 xs 200 applies, and `elt_df` lists the event once, with `units` 2. The mean annual loss is $0.1(500+100+200) = 80$.

```
s = build('''agg Shared elt unit Personal as "Personal lines" [[1 .1 300] [2 .1 100]]
                        elt unit Commercial [[1 .1 200] [3 .1 200]]
                        occurrence net of 300 xs 200''')
wprint(s.elt_description)
GT(s.elt_df, float_format=',.4g')
```

```
loss table              ELT3, 3 events from 4 rows, 2 units: Personal (Personal
                        lines), Commercial
event rate              0.3, mean annual loss 80
grid                    bs 100, log2 5
checks                  13 pass
```

event	rate	mean	units
1	0.1	500	2
2	0.1	100	1
3	0.1	200	1

Giving the Commercial row of event 1 a new identifier, 4, makes two occurrences, of 300 and 200, in place of one of 500. The gross mean stays 80, and the net mean moves from 50 to 70: the treaty cedes 100 of the 300 rather than 300 of the 500.

```
t = build('''agg Separate elt unit Personal [[1 .1 300] [2 .1 100]]
                          elt unit Commercial [[4 .1 200] [3 .1 200]]
                          occurrence net of 300 xs 200''')
print(f'{s.est_m:.2f} {t.est_m:.2f}')
```

```
50.00 70.00
```

Secondary uncertainty and the no2u flag

With secondary uncertainty, an ELT6 row makes X_e a beta distribution on $[0, M_e]$ with mean μ_e and SD $\sigma_I + \sigma_C$. When the variance $(\sigma_I + \sigma_C)^2$ is inadmissible for a law on $[0, M_e]$ with mean μ_e, that is at least $\mu_e(M_e - \mu_e)$, the row becomes the two-point law on $\{0, M_e\}$ with the same mean. The eltq package (`pip install aggregate[elt]`) computes the moments and quantizes the event laws on the grid. For example, the row `[1 .1 50 100 10 10]` defines a beta(2.625, 2.625) severity on $[0, 100]$, with annual mean 5 and variance $0.1(20^2 + 50^2) = 290$, an SD of 17.03.

```
b = build('agg Beta elt [[1 .1 50 100 10 10]]')
qd(b)
wprint(b.elt_description)
```

```
Aggregate object: Beta. Frequency distribution poisson. Severity elt,
[0, 100], bounded. Updated with bucket size 1/256 and log2 = 16.
Validation: not unreasonable.

      Mean      SD     CV       Skew    P01 Median    P99
X
Freq   0.1 0.31623 3.1623     3.1623
Sev     50      20    0.4 1.9824e-07 8.9336     50 91.066
Agg      5  17.029 3.4059     3.7458      0      0 80.055
loss table               ELT6, 1 event from 1 row, 1 unit
event rate               0.1, mean annual loss 5
secondary uncertainty    1 beta, 0 two-point, 0 atom rows
grid                     bs 1/256, log2 16, severity edge 255.998
checks                   19 pass
```

The severity functions of such an aggregate, `b.sev.sf`, `cdf`, `pdf`, `ppf`, and `isf`, are exact: eltq computes them from the rows, not from the grid.[2] The 99th percentile `b.sev.ppf(0.99)` of the beta(2.625, 2.625) law on $[0, 100]$ is 91.0665, and the quantile `q_sev(0.99)` on the grid, with bucket size 1/256, is 91.0664. Occurrence PMLs follow from them with `oep` (see Using oep).

```
print(f'{b.sev.ppf(0.99):.6f} {b.q_sev(0.99):.6f} {b.bs}')
```

```
91.066484 91.066406 0.00390625
```

The flag `no2u` (no secondary uncertainty) replaces each ELT6 row by an atom at its mean, which leaves the mean annual loss unchanged. The table below has one two-point row (event 2) and three betas. Secondary uncertainty doubles the SD, from 152 to 306, and nearly triples the 99.9% quantile, from 1,000 to 2,976.

[2] From aggregate 2.1.0a6

```
table = '''[[1 .10 300 3000 400 300] [2 .10 100 1000 371 267]
           [3 .25  50 1000 100 100] [4 .05 500 1000 120 100]]'''
for flag in ['no2u', '']:
    c = build(f'agg Cat elt {flag} {table}')
    print(f'{flag or "2U":5s} {c.est_m:8.2f} {c.est_sd:8.2f}',
          f'{c.q(0.99):8.0f} {c.q(0.999):8.0f}')
wprint(c.elt_description)
```

```
no2u     77.50   152.07      600     1000
2U       77.50   305.85     1473     2976
loss table               ELT6, 4 events from 4 rows, 1 unit
event rate               0.5, mean annual loss 77.5
secondary uncertainty    3 beta, 1 two-point, 0 atom rows
grid                     bs 1/2, log2 16, severity edge 32,768
checks                   19 pass
```

When an ELT6 event has rows in two units, the rows are rolled up into one before modeling: means, maximums and correlated SDs add, and independent SDs add in quadrature. In the aggregate `Rolled` below, event 1 has maximum $3{,}000 + 1{,}000 = 4{,}000$, independent SD $(400^2 + 300^2)^{1/2} = 500$ and correlated SD $300 + 100 = 400$.

```
r = build('''agg Rolled elt unit Personal [[1 .1 300 3000 400 300] [2 .1 100 1000 50 50]]
                        elt unit Commercial [[1 .1 200 1000 300 100]]''')
GT(r.elt_df, float_format=',.4g')
```

event	rate	mean	max	sdi	sdc	units
1	0.1	500	4,000	500	400	2
2	0.1	100	1,000	50	50	1

Refused programs

An event repeated within a unit, an event with different rates in different units, an ELT6 with secondary uncertainty beside a mean-only source, and a mean above its maximum each raise `LossTableError`, a subclass of `ValueError`. A row of the wrong length, and a loss table combined with `claims`, `sev`, `dsev`, or `approximate`, are refused when the program is parsed.

```
refuse('agg E1 elt [[1 .1 100] [1 .1 40]]',
       'agg E2 elt unit A [[1 .1 100]] elt unit B [[1 .2 40]]',
       'agg E3 elt [[1 .1 100 1000 10 10]] elt unit B [[2 .2 40]]',
       'agg E4 elt [[1 .1 100 50]]')
```

```
E1: an event appears more than once for one unit: elt source 1 (unit Total) row
1 (event 1), elt source 1 (unit Total) row 2 (event 1).
E2: an event's unit rows have different rates (an event has one rate, whatever
its units): elt source 2 (unit B) row 1 (event 1).
E3: an ELT6 source with secondary uncertainty cannot be combined with ELT3 or
ELT4 sources, including an ELT6 converted by no2u. Add no2u to elt source 1
(unit Total) for the mean-only model.
E4: elt source 1 (unit Total): mean exceeds max; see row 1 (id 1).
```

Using `elt` with files

Event loss tables read from CSV and Parquet files with the elt clause

Stephen J. Mildenhall

2026-10-30

An event loss table (ELT) from a catastrophe model usually arrives as a file. The `elt` clause takes a quoted path to one[1] in place of an inline table. The model is the one in Using elt: the annual loss is compound Poisson with expected count $\Lambda = \sum_e \lambda_e$, where λ_e is the rate of event e, and severity the mixture of the event losses with weights λ_e/Λ. Rows sharing an event identifier across units are one occurrence.

CSV, TSV, Parquet, and Feather files are read; the last two need `pyarrow`, and Excel is not read. Columns are matched by name after ignoring case, spaces, and underscores, through one alias table: `Event ID` is `event_id`, a rate may be `rate`, `frequency`, `freq`, or `annual_rate`, and the mean loss (`mean`, `mean_loss`, or `loss`) is the mean loss given the event, not the rate times it. A `unit` column splits one long file into units. The columns present set the form: ELT3 (identifier, rate, and mean loss), ELT4 (adding a maximum), or ELT6 (adding the independent and correlated standard deviations). Any other columns are ignored and listed. Identifiers are read as text, so they stay exact. A relative path resolves against the directory of the declaring `.agg` file, or the working directory when the statement is built.

The file is read once, when the aggregate is built, and its resolved path, size, modification time, and SHA-256 digest are recorded. Year loss tables (YELT) are read from files in the same way, as Using yelt shows.

```
agg Cat elt "cat.csv"                                    # units from a unit column
agg Split elt unit Personal "personal.parquet"
          elt unit Commercial "commercial.parquet"
agg Net elt "cat.csv" occurrence net of 300 xs 200
```

```
import io
import tempfile
from pathlib import Path
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.13.0

A long CSV file with a unit column

The CSV file below has one row per event and unit, headers as a vendor might write them, and an extra `Region` column. The code writes it to a temporary directory and builds the aggregate there, and the directory is removed when the `with` block ends.

```
csv = '''Event ID,Unit,Rate,Mean Loss,Max Loss,Region
1,Personal,0.10,300,3000,FL
2,Personal,0.10,100,1000,TX
1,Commercial,0.10,200,1000,FL
3,Commercial,0.25,50,1000,CA
'''
with tempfile.TemporaryDirectory() as d:
    path = Path(d, 'cat.csv')
    path.write_text(csv)
    a = build(f'agg CatFile elt "{path.as_posix()}" occurrence net of 300 xs 200')
qd(a)
```

```
Aggregate object: CatFile. Frequency distribution poisson. Severity
dhistogram, [50, 500], bounded; atoms [50 100 500]. Updated with bucket
size 50 and log2 = 6. Validation: reinsurance; subject not unreasonable.

       Mean      SD      CV    Skew P01 Median P99
X
Freq   0.45 0.67082  1.4907  1.4907
Sev  94.444  59.835 0.63355 0.99251  50     50 200
Agg    42.5      75  1.7647  2.2074   0      0 300
```

Event 1 has a row in each unit, so it is one occurrence of 500 with maximum 4,000. The table is ELT4, with a gross mean annual loss of $0.1(500+100)+0.25(50)=72.5$. The occurrence treaty 300 xs 200 cedes 300 of the 500, for a net mean of 42.5.

```
wprint(a.elt_description)
GT(a.elt_df, float_format=',.4g')
```

```
loss table              ELT4, 3 events from 4 rows, 2 units: Personal,
                        Commercial
event rate              0.45, mean annual loss 72.5
grid                    bs 50, log2 6
checks                  13 pass
```

event	rate	mean	max	units
1	0.1	500	4,000	2
2	0.1	100	1,000	1
3	0.25	50	1,000	1

Provenance of a file source

The second paragraph of `elt_explanation` lists the provenance of each source: the resolved path, size, modification time, the leading digits of the SHA-256 digest, and the ignored columns. The temporary directory is already gone, and the aggregate is unaffected.

```
wprint(a.elt_explanation.split('\n\n')[1])
print(path.exists(), f'{a.est_m:.4f}')
```

```
Source 1, unit Personal: file <tmp>/tmpu61cer9t/cat.csv (165 bytes, modified
2026-10-30T12:29:29, SHA-256 f3df7660ae67), ELT4, 2 rows, ignoring columns
Region. Source 1, unit Commercial: file <tmp>/tmpu61cer9t/cat.csv (165 bytes,
modified 2026-10-30T12:29:29, SHA-256 f3df7660ae67), ELT4, 2 rows, ignoring
columns Region.
False 42.5000
```

The stored program holds the absolute path of the file, so a rebuild reads the file again. With the directory gone, the rebuild fails.

```
wprint(a.program)
refuse(a.program)
```

```
agg CatFile elt "<tmp>/tmpu61cer9t/cat.csv" occurrence net of 300 xs 200
elt file "<tmp>/tmpu61cer9t/cat.csv": cannot read <tmp>\tmpu61cer9t\cat.csv
([Errno 2] No such file or directory:
'C:\\Users\\steve\\AppData\\Local\\Temp\\tmpu61cer9t\\cat.csv').
```

One Parquet file per unit

Two Parquet files, one per unit, can hold the same four rows as the CSV file of the first example, without its `Unit` column. With one `elt unit` clause per file they give the same aggregate, a net mean of 42.5. In the Parquet files pandas stores the identifiers as integers, and the clause converts each to its decimal text, so the events are `'1'`, `'2'`, and `'3'`, the same as from the CSV file.

```
cat = pd.read_csv(io.StringIO(csv))
with tempfile.TemporaryDirectory() as d:
    sources = []
    for unit, rows in cat.groupby('Unit', sort=False):
        rows.drop(columns='Unit').to_parquet(Path(d, f'{unit}.parquet'), index=False)
        sources.append(f'elt unit {unit} "{Path(d, unit).as_posix()}.parquet"')
    b = build(f'agg CatParquet {" ".join(sources)} occurrence net of 300 xs 200')
wprint(b.elt_description)
print(f'{a.est_m:.4f} {b.est_m:.4f}', list(b.elt_df.index))
```

```
loss table              ELT4, 3 events from 4 rows, 2 units: Personal,
                        Commercial
event rate              0.45, mean annual loss 72.5
grid                    bs 50, log2 6
checks                  13 pass
42.5000 42.5000 ['1', '2', '3']
```

Refused files

A Parquet identifier column of floating point numbers is refused, since its digits may already be lost; in a CSV file the same digits are text and are kept as written. Two columns with one meaning, an Excel file, and a `unit` clause that does not match the unit column of the file (the clause does not select rows) each raise `LossTableError`, a subclass of `ValueError`.

```
with tempfile.TemporaryDirectory() as d:
    q = lambda name: f'"{Path(d, name).as_posix()}"'
    floats = pd.DataFrame({'id': [1.0, 2.0], 'rate': [.1, .1], 'mean': [100, 50]})
    floats.to_parquet(Path(d, 'float.parquet'))
    Path(d, 'twice.csv').write_text('id,rate,freq,mean\n1,.1,.1,100\n')
    Path(d, 'cat.csv').write_text(csv)
```

```
    refuse(f'agg F1 elt {q("float.parquet")}',
           f'agg F2 elt {q("twice.csv")}',
           f'agg F3 elt {q("cat.xlsx")}',
           f'agg F4 elt unit Personal {q("cat.csv")}')
```

```
elt file "<tmp>/tmpui1_wi9_/float.parquet": the identifier column holds floating
point numbers, which may already have lost digits; store it as integers or text.
elt file "<tmp>/tmpui1_wi9_/twice.csv": columns 'rate' and 'freq' both mean
rate.
elt file "<tmp>/tmpui1_wi9_/cat.xlsx": unsupported file type '.xlsx'; use .csv,
.tsv, .parquet, .feather.
elt file "<tmp>/tmpui1_wi9_/cat.csv": the clause says unit Personal but the unit
column also holds Commercial.
```

Using `eventual_ruin`

Eventual ruin at a stated margin and capital with Aggregate.eventual_ruin

Stephen J. Mildenhall

2026-11-01

An insurer with initial surplus u, premium rate c, and claims X_i has surplus $U(t) = u+ct-\sum_{i\le N(t)} X_i$ at time t, and is ruined at the time $\tau = \inf\{t : U(t) < 0\}$. The probability of eventual ruin is $\psi(u) = \Pr(\tau < \infty)$. The premium rate is $c = (1+\rho)\mathsf{P}X/\mathsf{P}W$, where W is the wait between claims and the margin ρ is the ratio of premium margin to expected loss. The methods `pollaczeck_khinchine` and `wiener_hopf`, described in Using pollaczeck_khinchine and wiener_hopf, compute the whole function ψ. The method `Aggregate.eventual_ruin` evaluates it at a single surplus, and is used to find the ruin probability at a given capital or the capital at a given ruin probability.

The method `Aggregate.eventual_ruin`[1] takes the margin either as ρ or as the loss ratio $1/(1+\rho)$, and the capital either as a target ruin probability p or as a surplus u. It computes the exact $\psi(u)$, by `pollaczeck_khinchine` for a Poisson count and `wiener_hopf` for a renewal count, checks it by simulating surplus paths of the same discretized model, and returns a `RuinResult`. The simulation also records τ on each ruined path. These simulated times are the library's only information on the time to ruin; there is no finite-horizon ruin probability $\psi(u,t)$.

For exponential claims with mean m and Poisson arrivals at rate λ, the Lundberg exponent, or adjustment coefficient, is $R = \rho/((1+\rho)m)$, $\psi(u) = e^{-Ru}/(1+\rho)$, and the mean time to ruin given ruin is

$$\mathsf{P}(\tau \mid \tau < \infty) = \frac{u + (1+\rho)m}{\rho c}, \qquad c = (1+\rho)\lambda m.$$

Under the Lundberg conjugate measure ruin is certain and the claims exceed the premium at rate ρc, so by Wald's identity the mean ruin time under that measure is u plus the mean deficit at ruin, divided by ρc. The deficit is exponential with mean $(1+\rho)m$ and independent of τ, so the law of τ given ruin is the same under both measures. The examples use exponential claims with mean 1, and either a Poisson count of 10 claims a year, where these closed forms apply, or gamma waits with mean 2.

```
agg Po 10 claims sev expon poisson          # pollaczeck_khinchine, Lundberg rows
agg Ga 10 years sev expon wait gamma 2      # wiener_hopf
```

```
import numpy as np
from aggregate import build, qd, __version__
from aggregate.pedagogy import ruin_example
from aggregate.exhibits import exhibit_frames
from aggregate.charts import build_chart_doc
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.51.0
```

[1] From aggregate 1.0

Ruin probability at a stated margin and capital

For the Poisson aggregate at a margin of 20%, the call `eventual_ruin(rho=0.2, p=0.05)` finds the surplus with a 5% ruin probability, 16.87. The attribute `ruin_df` of the result is a one-column table of the margin, as ρ, as a loss ratio, and as a premium rate, the means of X and W, the resolved surplus, the exact and simulated ψ with the details of the simulation, and, for a Poisson count, the Lundberg exponent and the bound e^{-Ru}. With 1000 paths the simulated ψ, 0.062, is within two standard errors of the exact 0.050.

```
po = build('agg Po 10 claims sev expon poisson')
r = po.eventual_ruin(rho=0.2, p=0.05)
print(r.ruin_df)
```

```
                            value
frequency kind             poisson
safety loading rho             0.2
loss ratio                0.833333
premium rate c           11.999998
mean severity E[X]             1.0
mean wait E[W]                 0.1
initial surplus u        16.865234
psi(u) exact              0.049996
psi(u) simulated             0.062
sim std error             0.007626
sim trials                    1000
sim ruins                       62
sim horizon (claims)          1254
seed                      20260905
Lundberg exponent R       0.166667
Lundberg bound exp(-Ru)   0.060153
```

At the resolved surplus the closed form is within 2×10^{-4} of the computed ψ, which includes the discretization error of `pollaczeck_khinchine`. For exponential claims the Lundberg bound is exactly $(1+\rho)\psi(u)$.

```
R = 0.2 / 1.2
print(f'psi    closed form {np.exp(-R * r.u) / 1.2:.6f}   computed {r.psi:.6f}')
print(f'bound  closed form {np.exp(-R * r.u):.6f}   ruin_df  {r.ruin_df.value.iloc[-1]:.6f}')
```

```
psi    closed form 0.050127   computed 0.049996
bound  closed form 0.060152   ruin_df  0.060153
```

The simulation uses 1000 paths and a fixed seed by default, so a repeated call gives the same result. The argument `seed=None` draws a fresh seed and records it. The fields `rho`, `u`, `psi`, `psi_sim`, `se_sim`, and `seed` of the result contain the main values in `ruin_df`. The call below states the margin as a loss ratio of 0.8, so $\rho = 0.25$, and the capital as a surplus of 10, with ten times as many paths. The simulated ψ, 0.1115, is about one standard error from the exact 0.1081.

```
r2 = po.eventual_ruin(lr=0.8, u=10, n_sims=10_000)
print(f'rho {r2.rho:.4f}  u {r2.u}  psi {r2.psi:.6f}  '
      f'simulated {r2.psi_sim:.4f} ({r2.se_sim:.4f})  seed {r2.seed}')
```

```
rho 0.2500  u 10  psi 0.108057  simulated 0.1115 (0.0031)  seed 20260905
```

Ruin probability for a renewal count

For a renewal count, here gamma waits with mean 2, the method `eventual_ruin` uses `wiener_hopf`. Its `ruin_df` has no Lundberg rows. At a margin of 20% the surplus for a 1% ruin probability is 20.02, and 9 of the 1000 simulated paths are ruined.

```
ga = build('agg Ga 10 years sev expon wait gamma 2')
r3 = ga.eventual_ruin(rho=0.2, p=0.01)
print(r3.ruin_df)
```

```
                          value
frequency kind          renewal
safety loading rho          0.2
loss ratio             0.833333
premium rate c              0.6
mean severity E[X]          1.0
mean wait E[W]              2.0
initial surplus u     20.019531
psi(u) exact           0.009997
psi(u) simulated          0.009
sim std error          0.002986
sim trials                 1000
sim ruins                     9
sim horizon (claims)        973
seed                   20260905
```

Sample paths and the time to ruin

The `ruin` exhibit and the `ruin` chart present an `eventual_ruin` calculation in the aggregate Loss Lab, the browser application. The exhibit contains `ruin_df`, and the chart shows fifty sample paths and the ψ curve. The function `pedagogy.ruin_example` draws the same figure with matplotlib, from the same simulation, and returns a summary that includes the mean time to ruin given ruin.

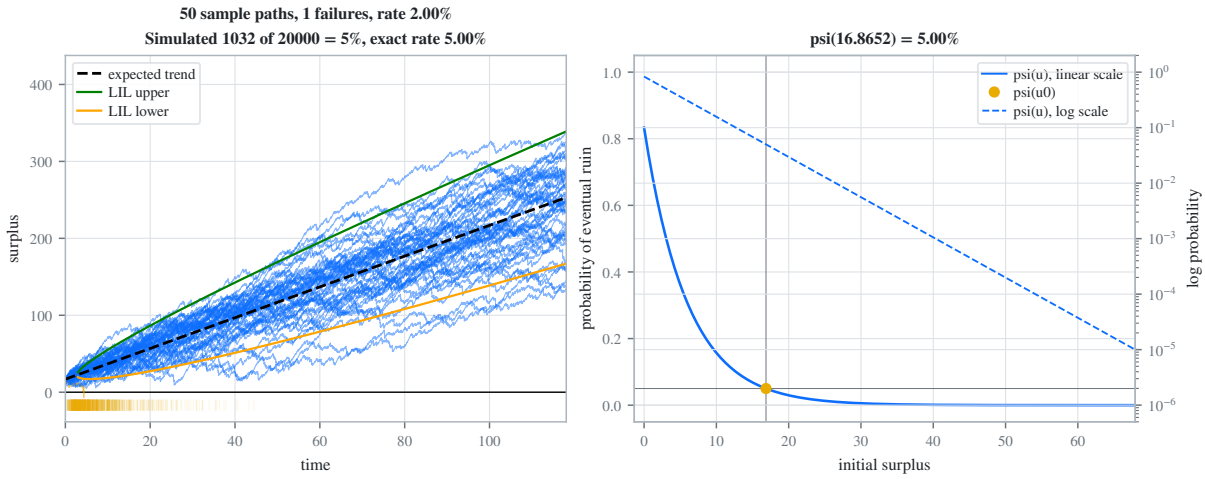
```
name, frame, meta = exhibit_frames(r, 'ruin')[0]
print(name, frame.equals(r.ruin_df))
wprint(meta['caption'])
doc = build_chart_doc(po, 'ruin', rho=0.2, p=0.05)
wprint(*[p.title for p in doc.panels], sep='\n')
```

```
ruin_df True
One eventual-ruin reading: the premium margin stated as safety loading, loss
ratio and premium rate, the model moments, the resolved initial surplus, and the
exact probability of eventual ruin there beside a simulated check with its
standard error. Poisson books add the Lundberg exponent and bound when the
adjustment equation has a root on the grid.
50 sample paths, 3 ruined; simulated 62 of 1000 = 6.2%, exact 5.0%
psi(16.8652) = 5.00%
```

The left panel of the figure shows sample paths with the expected trend, a band from the law of the iterated logarithm, and a rug of simulated ruin times. The right panel shows $\psi(u)$. Of 20,000 simulated paths 1,032 are ruined, and their mean time to ruin, 7.74, is about one standard error (some 0.21) above the closed form, 7.53.

```
summary, fig = ruin_example(po, 0.2, r.u, n_sims=20_000, seed=1)
fig.set_size_inches(9, 3.6)
rows = ['psi(u0) exact', 'psi(u0) simulated', 'sim std error', 'mean ruin time | ruin']
print(summary.loc[rows])
print(f'closed form mean time to ruin {(r.u + 1.2) / (0.2 * 12):.4f}')
```

```
                          value
psi(u0) exact          0.049996
psi(u0) simulated        0.0516
sim std error          0.001564
mean ruin time | ruin  7.739487
closed form mean time to ruin 7.5272
```



Refused calls

The method `eventual_ruin` requires exactly one positive margin, as `rho` or as a loss ratio `lr` below 1, and exactly one capital, as `p` or as a surplus `u` within the grid. The count must be Poisson or renewal.

```
nb = build('agg Nb 10 claims sev expon negbin 0.5')
refuse(lambda: po.eventual_ruin(rho=0, p=0.05),
       lambda: po.eventual_ruin(rho=0.2, lr=0.8, p=0.05),
       lambda: po.eventual_ruin(rho=0.2),
       lambda: po.eventual_ruin(lr=1.2, p=0.05),
       lambda: po.eventual_ruin(rho=0.2, u=1000),
       lambda: nb.eventual_ruin(rho=0.2, p=0.05))
```

```
net profit condition requires rho > 0, got 0
exactly one of rho and lr is required
exactly one of p and u is required
loss ratio must be in (0, 1), got 1.2
u0 = 1000 lies beyond the represented grid top 127.998
no eventual-ruin solver for a 'negbin' frequency: need poisson
(pollaczeck_khinchine) or renewal (wiener_hopf)
```

Using explanations

The description and explanation strings on aggregate objects

Stephen J. Mildenhall

2026-11-01

Many facts about a computed object are reported twice, once in a table and once in words: whether the grid is good enough, why the bucket has its size, how thick the tail is, and what the reinsurance cedes. The words come as pairs of read-only string properties sharing a stem[1]. The string `<stem>_description` is one line or a few aligned lines, for scanning. The string `<stem>_explanation` is prose: what was checked or chosen, against what, and what to do about it. Wherever one half exists the other does too. Both are computed on access, so they follow the object as it changes. The short forms of validation, tail, and reinsurance also appear in the attribute `info`.

Validation compares the severity and aggregate mean, coefficient of variation (CV), and skewness on the realized grid with their analytic values. A relative error above `validation_eps` (10^{-4} by default) fails a mean, above ten times that a CV, and above a hundred times that a skewness. For the severity and for the aggregate only the lowest-order failure is reported, since a wrong mean makes the higher moments uninformative.

The examples below use objects built from these programs.

```
agg Tour 20 claims sev lognorm 100 cv 1.5 occurrence net of 500 xs 100 poisson
port Two
  agg A 10 claims sev gamma 50 cv 0.5 poisson
  agg B 5 claims sev lognorm 100 cv 1 poisson
agg LT.Basic elt [[1 .1 100] [2 .1 40] [3 .1 80]]
pnl PL 100 premium less agg PL.B 10 claims sev lognorm 8 cv 1.5 poisson
```

```
import pandas as pd
from aggregate import build, qd, explain_validation, __version__
from aggregate import (Aggregate, Portfolio, PnL, Severity, Frequency,
                       Distortion)
from aggregate.bivariate import BivariateAggregate
from style import wprint, GT, figures
figures()
print(__version__)
```

```
1.58.0
```

Description and explanation pairs by class

The table marks, for each class, the stems that have a pair. It is computed from the classes themselves, so it stays current. A `Distortion` has none of the pairs.

[1] From aggregate 1.0, and the `elt` pair from 1.21.0

```
stems = ['validation', 'bs', 'tail', 'sharpen', 'reins', 'elt', 'construction']
classes = [Aggregate, Portfolio, BivariateAggregate, PnL, Severity, Frequency,
           Distortion]
has = {s: ['x' if f'{s}_description' in dir(c) else '' for c in classes]
       for s in stems}
names = pd.Index([c.__name__ for c in classes], name='class')
GT(pd.DataFrame(has, index=names))
```

class	validation	bs	tail	sharpen	reins	elt	construction
Aggregate	x	x	x	x	x	x	
Portfolio	x	x	x	x	x		
BivariateAggregate	x	x	x				
PnL							x
Severity			x				
Frequency			x				
Distortion							

Validation against the analytic moments

The string `validation_description` is the verdict. Under reinsurance it has two clauses, one for the gross subject against its analytic moments and one for the net against the exact image of the subject on the grid (Using validation). The function `explain_validation` returns the words for the flags in the attribute `valid`. Flags alone cannot separate a subject from a view, so on a reinsured object it gives only the first clause.

```
a = build('agg Tour 20 claims sev lognorm 100 cv 1.5 '
          'occurrence net of 500 xs 100 poisson')
print(a.validation_description)
print(explain_validation(a.valid))
wprint(a.validation_explanation)
```

```
reinsurance; subject not unreasonable; net not unreasonable
reinsurance; subject not unreasonable
Validation of the realized grid against the analytic moments: reinsurance;
subject not unreasonable; net not unreasonable. Each of mean, CV and skewness is
compared for severity and for the aggregate, and a relative error above 0.0001
fails; only the lowest-order failure is reported, since a mean that is wrong
makes the higher moments uninformative. The reported view is net or ceded. The
subject (gross) object underneath is checked against its analytic moments, and
the cession against the exact image of the subject on its grid.
```

The choice of bucket and window

The string `bs_description` gives the grid and any clipped tail, and `bs_explanation` gives the sizing methods that applied, the winner, and any single big jump reach the window misses. The table behind them is `bs_window_df`, described in Using FFT windows and, with the tail classes, in Using tail classes.

```
wprint(a.bs_description)
wprint(a.bs_explanation)
```

```
moment grid: bs=1/2, log2=16, window [0, 32,768] of width 32,768; clips
~5.79e-08 of the tail (raise log2 to 19)
The aggregate is severity-driven, [0, inf), subexponential right tail. Log2 is
16. Of the methods that applied (moment 29,492, sbj 180,738), the moment method
won (window width 29,492) -- the 3-moment method-of-moments window. It has a
natural lower support bound of 0. The recommended x_min is 0 and the x_max that
follows is 32,768, so the window is [0, 32,768] of width 32,768. The heavy right
tail reaches 180,738, past the grid x_max 32,768 (~5.79e-08 of the mass clipped,
a reported deficit not normalized); the analysis suggests increasing log2 to 19.
```

Reinsurance terms and cessions

The string `reins_description` restates the reinsurance terms, and `reins_explanation` adds their effect on expected loss, stage by stage. For `Tour` the occurrence layer 500 xs 100 cedes 727 of the gross expected loss of 2,000, or 36.4 percent.

```
wprint(a.reins_description)
wprint(a.reins_explanation)
```

```
Net of 100% share of 500 xs 100 per occurrence
Net of 100% share of 500 xs 100 per occurrence. Per occurrence, gross expected
loss 2,000 splits into 727.203 ceded (36.4%) and 1,272.8 net.
```

The sharpen pair

Before the method `sharpen()` is called, the string `sharpen_description` is `Sharpen: not run.` After the call it gives the probe and its outcome (Using `sharpen`). The explanation runs to a page, with the definition of the score and of the search. It ends with the reason the grid moved or stayed, the part shown here.

```
print(a.sharpen_description)
a.sharpen()
wprint(a.sharpen_description)
wprint(a.sharpen_explanation.rsplit('Here: ', 1)[1])
```

```
Sharpen: not run.
Sharpen: 6 cells in 0.14s, best score 0.285 vs 0.285 at the center, target 0.5.
Kept bs 1/2, log2 16.
best sound score at the current grid size or smaller, and it meets the target,
so log2 was not grown. No neighbor was enough better to justify moving, so the
grid was kept.
```

Severity and frequency tails

A `Severity` and a `Frequency` have only the tail pair. The components of an aggregate are available as `sevs`, a list of severities, and `frequency`.

```
wprint(a.sevs[0].tail_explanation)
wprint(a.frequency.tail_explanation)
```

```
Lognorm severity [0, inf), subexponential right tail. Declared support:
unlimited. The right tail is thick (subexponential): the aggregate inherits it
by the single big jump.
Poisson frequency with a super-exponential count tail. E[N] = 20, SD(N) =
4.47214. It is Poisson-dispersed (var / mean = 1). The count law is log-concave,
so the aggregate tail follows the severity.
```

The tail and grid pairs of a Portfolio

On a `Portfolio` the tail pair gives the unit that drives the tail of the total, and the grid pair describes the one grid all the units share. In `Two` the lognormal unit `B` drives the subexponential tail.

```
p = build('port Two agg A 10 claims sev gamma 50 cv 0.5 poisson '
          'agg B 5 claims sev lognorm 100 cv 1 poisson')
wprint(p.tail_description)
wprint(p.tail_explanation)
wprint(p.bs_description)
```

```
aggregate tail           subexponential (driver: B)
Portfolio aggregate tail is subexponential (worst-of under independence, driven
by B). Units -- A: exponential; B: subexponential.
portfolio grid: bs=1/2, log2=16, window [0, 32,768] of width 32,768
```

The elt pair for loss tables

An `elt` or `yelt` aggregate (Using `elt`, Using `yelt`) has the `elt` pair. The description is a few aligned lines with the table, the event rate, the grid, and the checks. The explanation is three paragraphs, the first of which describes the model in words. On any other aggregate both strings are empty.

```
e = build('agg LT.Basic elt [[1 .1 100] [2 .1 40] [3 .1 80]]')
wprint(e.elt_description)
wprint(e.elt_explanation.split('\n\n')[0])
print(repr(a.elt_description), repr(a.elt_explanation))
```

```
loss table               ELT3, 3 events from 3 rows, 1 unit
event rate               0.3, mean annual loss 22
grid                     bs 20, log2 5
checks                   13 pass
The annual loss is compound Poisson with event rate 0.3 and severity the
rate-weighted mixture of the event totals, so the mean annual loss is 22. An
event's unit losses are summed into one event total before any occurrence
treaty, and equal totals are combined: the 3 events with a positive rate give 3
severity atoms.
'' ''
```

The construction pair of a PnL

A `PnL` (Using `pnl` and `xpnl`) has the `construction` pair. The description is one paragraph with the route, the source, and the groups. The explanation opens with the description and adds the booking and a replay of the constructor call, with placeholders for function legs.

```
q = build('pnl PL 100 premium less '
          'agg PL.B 10 claims sev lognorm 8 cv 1.5 poisson')
wprint(q.construction_explanation)
```

```
Plain pnl, consolidated face: one sell group over the engine's own density
(Aggregate); consideration 100; scenario (κ) ladder (single shared source).
Booking: sell -- +consideration, -obligation; loss-basis LAE is stochastic rate
* x (the loss is on-source). For a reinsurance-bearing engine the density here
is the net -- what comes out of the engine.

Replay (function legs as <fn> placeholders):
    PnL(name='PL',
        source=<GridDistribution PL>,
        groups=[Group('PL', 'sell', consideration=[Leg('Premium', 100.0,
        kind='premium')], obligation=[Leg('Loss', <fn>, kind='loss')])],
        result_name='margin')
```

Using FFT windows

The FFT grid window chosen by the bucket sizer, shown by bs_window_df

Stephen J. Mildenhall

2026-10-30

An aggregate is computed on $n = 2^\ell$ buckets of size b, at the points $x_k = x_0 + kb$ for $k = 0, \dots, n-1$, so on the **window** $[x_0, x_0 + W)$ of width $W = nb$. In code the bucket size is `bs` and ℓ is `log2`. Convolution by FFT is circular: it returns the law of S reduced modulo W. Provided one severity fits in $[0, W)$, any window of width W that holds essentially all the mass therefore gives the right answer once the result is relabeled onto it, and the origin x_0 is free.

The bucket sizer uses that freedom. For an ordinary non-negative law $x_0 = 0$. A **concentrated** law, one whose coefficient of variation (CV) is below 0.1, has its mass far above zero, and the window is placed around it, so the same n gives a finer bucket. For a signed severity the window straddles zero and $x_0 < 0$. Nothing is declared: the sizer selects the window and `density_df` is indexed on it. The table `bs_window_df` lists the candidate windows, one row per sizing method, with the width `W` of each and the probability `coverage` it is sized to hold.

A concentrated law is computed on a window away from zero[1]. The attributes `bs_window_df`, `bs_description`, and `bs_explanation` describe the choice of window. The tail class of the severity also enters that choice, as described in Using tail classes. The examples are three aggregates, one of each kind.

```
agg Ord    10 claims sev gamma 50 cv 0.5 poisson       # window [0, W)
agg Conc   10000 claims sev gamma 10 cv 0.5 poisson    # concentrated, far from 0
agg Signed 10 claims ssev 50 * norm + 10 poisson       # signed, straddles 0
```

```
from aggregate import build, __version__
from style import wprint, GT, figures
figures()
print(__version__)
```

```
1.43.1
```

The window of an ordinary aggregate

For a non-negative aggregate that is not concentrated the window starts at 0. Each row of `bs_window_df` is a window $[x_{min}, x_{max}]$ with its bucket. The `moment` row is the window of a three-moment fit, `sbj` is the single big jump floor on the window, described in Using tail classes, and `used` is the realized window, of width $2^{16}/16 = 4{,}096$. The `selected` column marks the method whose bucket was taken.

[1] From aggregate 1.0

```
cols = ['applies', 'selected', 'x_min', 'x_max', 'W', 'bs', 'log2', 'coverage']
fmt = {'x_min': ',.0f', 'x_max': ',.0f', 'W': ',.0f', 'bs': '.4g'}
a = build('agg Ord 10 claims sev gamma 50 cv 0.5 poisson')
wprint(a.bs_description)
GT(a.bs_window_df[cols], formatters=fmt)
```

```
moment grid: bs=1/16, log2=16, window [0, 4,096] of width 4,096
```

method	applies	selected	x_min	x_max	W	bs	log2	coverage
moment	True	True	0	2,573	2,573	0.0625	16	1-1e-12
sbj	True	False	0	1,166	1,166	0.03125	16	E[N]-adj 1-1e-12
used	True	False	0	4,096	4,096	0.0625	16	1-1e-12

The window of a concentrated aggregate

The aggregate `Conc` has ten thousand claims, mean 100,000, and CV 0.011. The `moment` window $[0, 107{,}987]$ would need $b = 2$. The `windowed` method holds the mass in a band of width 13,248 and takes $b = 1/4$, eight times finer on the same n. The `used` row is that band padded out to 2^{16} buckets, with the slack split on either side. The coverage differs by edge. For a loss the right tail, which determines the risk measures, is covered to $1 - 10^{-12}$, and the left only to $1 - 10^{-6}$.

```
c = build('agg Conc 10000 claims sev gamma 10 cv 0.5 poisson')
wprint(c.bs_description)
GT(c.bs_window_df[cols], formatters=fmt)
```

```
windowed grid: bs=1/4, log2=16, window [93,484, 109,868] of width 16,384
```

method	applies	selected	x_min	x_max	W	bs	log2	coverage
moment	True	False	0	107,987	107,987	2	16	1-1e-12
windowed	True	True	94,739	107,987	13,248	0.25	16	lo 1-1e-6 / hi 1-1e-12
sbj	True	False	0	100,148	100,148	2	16	E[N]-adj 1-1e-12
used	True	False	93,484	109,868	16,384	0.25	16	lo 1-1e-6 / hi 1-1e-12

For a concentrated aggregate `density_df` and `xs` start at the window, not at 0, and quantiles and moments are computed from them as usual. The first buckets hold probabilities of order 10^{-12}. The median is 99,997.5, the mean is 100,000, and the aggregate passes validation.

```
print(c.xs[0], c.xs[-1], len(c.xs))
print(c.density_df[['p_total', 'F']].head(3))
print(c.q(0.5), c.est_m, c.validation_description)
```

```
93484.5 109868.25 65536
              p_total             F
loss
93484.50  2.478809e-12  2.478809e-12
93484.75  2.482170e-12  4.960979e-12
93485.00  2.485535e-12  7.446514e-12
99997.5 100000.00003263868 not unreasonable
```

The string `bs_explanation` gives the same choice in prose. It lists the width of each candidate window, the winning method, and the recommended window.

```
wprint(c.bs_explanation)
```

```
The aggregate is severity-driven, [0, inf), exponential right tail. Log2 is 16.
Of the methods that applied (moment 107,987, windowed 13,248, sbj 100,148), the
windowed method won (window width 13,248) -- a two-sided window far from 0
(benign FFT wrap). It has a natural lower support bound of 0. The distribution
is concentrated with a CV of 0.0111803. The recommended x_min is 93,484 and the
x_max that follows is 109,868, so the window is [93,484, 109,868] of width
16,384.
```

Passing `x_min=0` to `update` forces a window starting at 0. The sizer then takes the coarse `moment` bucket, $b = 2$, and the severity and aggregate CVs fail validation.

```
c0 = build('agg Conc 10000 claims sev gamma 10 cv 0.5 poisson')
c0.update(x_min=0)
wprint(c0.bs_description)
print(c0.validation_description)
```

```
moment grid: bs=2, log2=16, window [0, 131,072] of width 131,072
fails sev cv, agg cv
```

The window of a signed severity

An `ssev` severity takes both signs (Using `ssev`), so the `moment` window is two-sided and starts below zero, at -826.75. The `used` row keeps that left edge and pads the window to 2^{16} buckets of size $1/16$ on the right.

```
s = build('agg Signed 10 claims ssev 50 * norm + 10 poisson')
wprint(s.bs_description)
display(GT(s.bs_window_df[cols], formatters=fmt))
print(s.xs[0], s.xs[-1])
```

```
moment grid: bs=1/16, log2=16, window [-826.75, 3,269] of width 4,096
```

method	applies	selected	x_min	x_max	W	bs	log2	coverage
moment	True	True	-827	1,504	2,331	0.0625	16	1-1e-12
sbj	True	False	-267	467	735	0.01562	16	E[N]-adj 1-1e-12
used	True	False	-827	3,269	4,096	0.0625	16	1-1e-12

```
-826.75 3269.1875
```

cdf and sf outside the window

Below the window of a non-negative law that starts at 0, the method `cdf` returns 0 and the method `sf` returns 1. Below a window placed above the origin, or below the window of a signed law, the mass there was trimmed or never computed, so both are `nan`. Above the window both come from the mass on the grid, so far above it `sf` returns the shortfall. For the aggregates `Conc` and `Signed` the shortfall, 1.8×10^{-9} and 6.2×10^{-8}, is the left tail below the window. For `Signed` it exceeds the 10^{-12} in the `coverage` column, which is the target of the three-moment fit, not a measured probability.

```python
for ob, lo in [(a, -1), (c, 1000), (s, -2000)]:
    print(f'{ob.name:7s} cdf({lo}) = {ob.cdf(lo)}, sf({lo}) = {ob.sf(lo)}, '
          f'sf(1e6) = {ob.sf(1e6):.2e}')
```

```
Ord     cdf(-1) = 0.0, sf(-1) = 1.0, sf(1e6) = 2.12e-14
Conc    cdf(1000) = nan, sf(1000) = nan, sf(1e6) = 1.78e-09
Signed  cdf(-2000) = nan, sf(-2000) = nan, sf(1e6) = 6.22e-08
```

Using `grossnet`, `grossceded` and `netceded`

Joint distribution of gross, ceded and net loss under occurrence reinsurance with the view-pair prefixes

Stephen J. Mildenhall

2026-10-30

An aggregate with occurrence reinsurance has three views of the annual loss: gross G, ceded C, and net $N = G - C$. The `agg` computes the distribution of each, but the three marginal distributions do not determine how the views move together. The recovery in a bad gross year depends on whether the year is bad because of many small claims or a few large ones. The DecL prefixes `netceded`, `grossceded`, and `grossnet` turn an `agg` with occurrence reinsurance into a `BivariateAggregate` of two of its views, x then y in the order the keyword lists them: `netceded` is (net, ceded), `grossceded` is (gross, ceded), and `grossnet` is (gross, net).

The joint law is built claim by claim. A gross claim x cedes $c(x)$ and retains $x - c(x)$, so the per-claim pair lies on the graph of the cession function. The pair is compounded with the shared claim count by a two-dimensional FFT, as in Using bivariate. Each marginal reproduces the corresponding view of the `agg`, and in `netceded` the total $x + y$ is the gross loss. The expected recovery given the gross outcome, $\kappa(g) = \mathsf{P}(C \mid G = g)$, is computed at every gross outcome by `exeqa_df`, and `conditional` returns the whole conditional law at one outcome.

The prefix goes in front of an `agg` program, written in full or as a reference to a stored aggregate. Only the occurrence program is used: annual terms such as `aad` and `agglimit`, and economic clauses such as `rate` and `cede`, are ignored with a warning, because the joint is of losses against losses. From Python, `Aggregate.occ_bivariate(views=('gross', 'ceded'))` returns the same object from a built aggregate.

```
grossceded agg Book 10 claims 1000 xs 0 sev lognorm 50 cv 2 occurrence net of 750 xs 250 poisson
grossnet agg GrossNet agg.Book          # by reference to a stored aggregate
netceded agg NetCeded agg.Book
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
1.43.1
```

Three view pairs of one program

The example book has ten claims a year from a lognormal with mean 50 and coefficient of variation 2, limited to 1,000, and cedes everything above 250 on each occurrence. The `agg` is built and stored first, with mean gross, ceded, and net losses 492.65, 52.81, and 439.84. Each view pair is then built by reference to it.

```
a = build('agg Book 10 claims 1000 xs 0 sev lognorm 50 cv 2 '
          'occurrence net of 750 xs 250 poisson')
print(a.reins_stats_df.loc[('agg', 'mean')])
```

```
view  layer
occ   Gross      492.651736
      layer.1     52.811908
      Ceded       52.811908
      Net        439.839828
Name: (agg, mean), dtype: float64
```

The table lists the two axes of each pair, their means, the mean of $x + y$, and the correlation of x and y. The marginal means match the `agg` in every pair. In `netceded` the total is the gross loss, 492.65, while in the other two pairs the total has no direct meaning. Gross and net are the most closely related pair, with correlation 0.904, and net and ceded the least, with 0.403.

```
pairs = {kw: build(f'{kw} agg {kw.title()} agg.Book')
         for kw in ['netceded', 'grossceded', 'grossnet']}
df = pd.DataFrame.from_dict(
    {kw: [*b.unit_names, *b.summary_df.Mean, b.corr] for kw, b in pairs.items()},
    orient='index', columns=['x', 'y', 'mean x', 'mean y', 'mean x + y', 'corr'])
df.index.name = 'prefix'
GT(df, float_format=',.2f', formatters={'corr': '{x:.3f}'})
```

prefix	x	y	mean x	mean y	mean x + y	corr
netceded	Net	Ceded	439.84	52.81	492.65	0.403
grossceded	Gross	Ceded	492.65	52.81	545.46	0.753
grossnet	Gross	Net	492.65	439.84	932.49	0.904

Recovery given the gross outcome

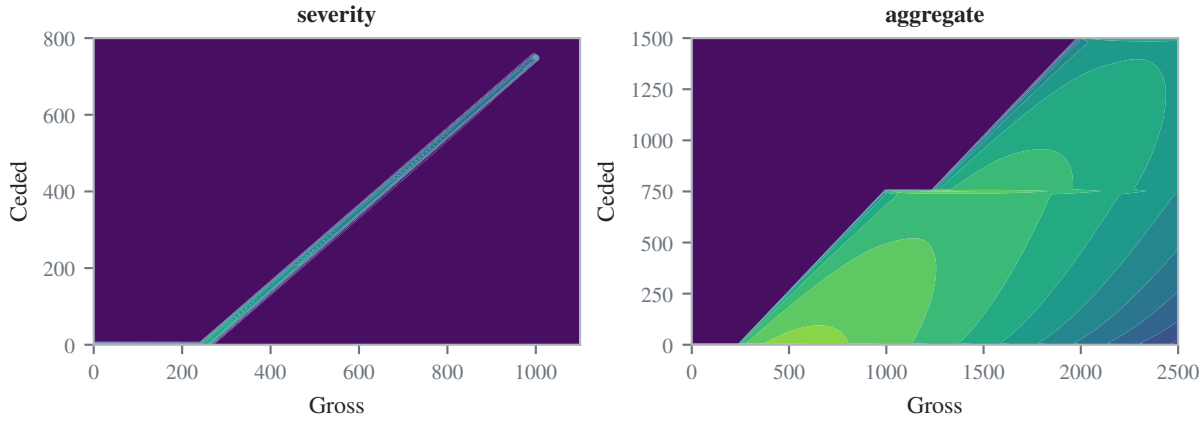
The pair `grossceded` gives the conditional law of the recovery at each gross outcome. The table shows it at four gross outcomes: the median and the 1 in 10, 1 in 100, and 1 in 1,000 years. Over all years the treaty cedes 52.81/492.65 = 10.7% of gross loss, but the share is far from constant. In a median year, 424, the expected recovery is 3.7 and 92% of such years recover nothing. In the 1 in 10 year, 904, the expected recovery is 153, or 16.9% of gross, yet 19% of those years recover nothing, and the 1st to 99th percentile range runs from 0 to 528. In the 1 in 100 year, 1,544, the share is 39.8%, and nearly every such year recovers something.

```
gc = pairs['grossceded']
gross = gc.marginal('Gross')
rows = {}
for year, p in {'median': 0.5, '1 in 10': 0.9, '1 in 100': 0.99,
                '1 in 1,000': 0.999}.items():
    g = gross.q(p)
    c = gc.conditional('x', g)
    rows[year] = [g, c.mean(), c.mean() / g, c.cdf(0), c.q(0.01), c.q(0.99)]
df = pd.DataFrame.from_dict(rows, orient='index', columns=[
    'gross', 'mean ceded', 'share', 'Pr(no recovery)', 'q01', 'q99'])
df.index.name = 'gross year'
GT(df, float_format=',.1f', ratio_cols=['share'],
   formatters={'Pr(no recovery)': '{x:.4f}'})
```

gross *year*	gross	mean ceded	share	Pr(no recovery)	q01	q99
median	424	3.7	0.9%	0.9180	0	88
1 in 10	904	153.0	16.9%	0.1922	0	528
1 in 100	1,544	613.8	39.8%	0.0015	88	920
1 in 1,000	2,104	958.9	45.6%	0.0000	392	1,496

The method `plot` draws the per-claim pair on the left and the joint aggregate on the right, here with a log color scale and the axes cut to the bulk of the law. The per-claim pair is a line: nothing is ceded below 250 and every dollar above 250 is ceded. In the aggregate the recovery lies between zero and gross less 250, and the ridge at 750 holds the years whose only claim above 250 is at the 1,000 limit.

```
fig, (ax0, ax1) = figure('full', cols=2)
gc.plot(axs=(ax0, ax1), log=True)
ax0.set(xlim=[0, 1100], ylim=[0, 800])
ax1.set(xlim=[0, 2500], ylim=[0, 1500]);
```



Bad gross years and bad net years

The pair `grossnet` shows how often a bad gross year is also a bad net year, which the two marginal distributions cannot show. Gross exceeds its 1 in 100 level, 1,544, in 1% of years, and in 36.1% of those years net exceeds its own 1 in 100 level, 1,080. The probability would be 1% if gross and net were independent and 100% if they were comonotone. Net loss is the sum of the claims capped at 250, so a net loss above 1,080 needs at least five claims, and a bad gross year made by one or two large claims is a moderate net year.

```
gn = pairs['grossnet']
qg, qn = gn.marginal('Gross').q(0.99), gn.marginal('Net').q(0.99)
d = gn.density_df
bad_gross = d.loc[d.index > qg].values.sum()
both = d.loc[d.index > qg, d.columns > qn].values.sum()
print(qg, qn, bad_gross, both / bad_gross)
```

```
1544.0 1080.0 0.009994323856094145 0.3608951908764983
```

Refused programs

A view pair needs occurrence reinsurance to split, so a prefix on a plain `agg` is refused.

```
refuse('netceded agg Plain 10 claims 1000 xs 0 sev lognorm 50 cv 2 poisson')
```

```
netceded requires occurrence reinsurance; none configured.
```

Using help

Help in aggregate with decl_help, the help method, and parse error reports

Stephen J. Mildenhall

2026-11-01

The package `aggregate` offers help at three levels: for the DecL language, for an object, and for a mistake. The function `decl_help()`[1] returns a page of DecL help as HTML. The method `.help(regex)`[2] lists the public methods and properties of an object whose names match a regular expression. A DecL parse error comes with a structured report[3]: where the error is, what would have been accepted there, a "did you mean" suggestion, and, for a known trap, a hint with the rule that was broken.

```
decl_help()                    # DecL at a glance: one agg taken apart slot by slot
decl_help('card')              # DecL on one page: a ten box reference card
a.help('^est_')                # public names of a that match a regex
a.help('^q$', lod='short')     # with the first lines of each docstring
e.report                       # the ErrorReport on a parse error e
```

```
import re, html
from aggregate import build, qd, decl_help, DECL_HELP_FORMS, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.60.0
```

DecL help pages

The DecL help pages are listed in `DECL_HELP_FORMS`. The first, `anatomy`, is the default and shows one `agg` statement slot by slot; the second, `card`, is a reference card of ten boxes. A page is a `str` containing one self-contained HTML `<section>`. As the last expression of a notebook cell a page is rendered as HTML, through its `_repr_html_`, and `IPython.display.HTML(page)` renders it too. Elsewhere its repr shows the form, the version, and the size, rather than the HTML.

```
print(DECL_HELP_FORMS)
page = decl_help()
page
```

```
('anatomy', 'card')
```

[1] From aggregate 1.2.0
[2] From aggregate 1.0
[3] From aggregate 1.0

`agg Book`	label	keyword, then a label you choose
`250 claims` `1000 xs 0`	exposure	**volume**: amount of business optional **coverage**: limit xs deductible
`sev lognorm 100 cv 1.5`	severity	severity curve: family, mean, cv
`occurrence net of 250 xs 750`	occurrence reinsurance	optional, before frequency
`poisson`	frequency	claim count distribution
`aggregate net of 5000 xs 30000`	aggregate reinsurance	optional, after frequency
`note{main book}`	trailer	optional note
`hints{bs=1;log2=16;}`	trailer	optional update hints

Exposure volume	**coverage (optional)**	**Severity**	**Frequency**	**Reinsurance**

`250 claims` expected count
`25000 loss` expected loss
`40000 premium at 65% lr`
`1000 exposure at 25 rate`
units × rate
	`1000 xs 0` limit xs deductible
	omit for unlimited
	`[500 1000] xs [0 0]` a mix
		`sev lognorm 100 cv 1.5` mean, cv
		`sev gamma 100 cv 1` any scipy.stats
		`sev 50 * invgamma 2.5` scale, shape
		`sev ... !` unconditional
			`poisson`
			`mixed gamma 0.3` mixing cv 0.3
			`fixed` exactly the count
				`occurrence net of 75% po 600 xs 400` partial placement
				`aggregate ceded to 250 xs 250 and 500 xs 500` limit xs attachment, lowest first
				`aggregate net of 66% po inf xs 0 rate 100% cede 30%` 66% quota share w 30% ceding commission

less common				
`[100 200 50] claims` vectorized exposure `dfreq [1 2 3] [.5 .3 .2]` listed counts and probabilities (optional, default equally likely)	`4000 xs 1000` an excess layer	`dsev [1 2 10] [.4 .4 .2]` listed `sev 40 * expon + 10` scale and shift `picks [250 1000] [20000 4500]` adjust expected loss by layer	`negbin 2` variance multiple `binomial 0.5`	`50% po 500 xs 500` partial placement `tower [0 250 500 1000]` program breakpoints `rol 5% deposit 600 reinstatements 1 at 100%`

A help page is an ordinary string, so the usual string tools work on it. The regular expression below

extracts the titles of the ten boxes on the card.

```
card = decl_help('card')
print(repr(card))
wprint(re.findall(r'<h3[^>]*>(.*?)</h3>', card))
```

```
DeclHelp('card', aggregate 1.60.0, 23 kB)
['1 · An aggregate', '2 · Exposure', '3 · Severity', '4 · Frequency', '5 ·
Reinsurance', '6 · Reinsurance terms', '7 · A portfolio', '8 · A profit and
loss', '9 · Writing rules', '10 · The library, and more']
```

Every program on a help page builds, and the library's tests check that it does. A program that a host, such as the aggregate Loss Lab, can load is stored in a `data-load` attribute. The first on the anatomy page is built below.

```
programs = [html.unescape(p) for p in re.findall(r'data-load="([^"]*)"', page)]
print(len(programs))
print(programs[0])
qd(build(programs[0]))
```

```
3
agg Book
  250 claims
  1000 xs 0
  sev lognorm 100 cv 1.5
  occurrence net of 250 xs 750
  poisson
  aggregate net of 5000 xs 30000
  note{main book}
  hints{bs=1;log2=16;}
Aggregate object: Book. Frequency distribution poisson. Severity
lognorm, [0, 1,000], bounded; 1,000 xs 0. Updated with bucket size 1 and
log2 = 16. Validation: reinsurance; subject not unreasonable; net not
unreasonable.

       Mean     SD       CV     Skew   P01 Median   P99
X
Freq    250 15.811 0.063246 0.063246
Sev  96.712  120.2   1.2428    2.926     4     55   693
Agg   24167   2408 0.099643 0.085218 18823  24107 30000
```

The help method

The help method `.help(regex)` is available on `Aggregate`, `Portfolio`, `Severity`, `Frequency`, `Distortion`, `PnL`, `Copula`, `BivariateAggregate`, `Bounds`, `GridDistribution`, and `Underwriter`. It lists the public names that match `regex`, using `re.search`, and gives each method with its signature. With no argument the regex is `'.*'`, which matches every public name. The code below counts the public names of a dice aggregate and then lists those that match `^est_`.

```
a = build('agg Dice dfreq [3] dsev [1:6]')
print(len([n for n in dir(a) if not n.startswith('_')]))
a.help('^est_')
```

```
175
Attribute: est_cv
Attribute: est_m
Attribute: est_sd
```

```
Attribute: est_sev_cv
Attribute: est_sev_m
Attribute: est_sev_sd
Attribute: est_sev_skew
Attribute: est_sev_var
Attribute: est_skew
Attribute: est_var
```

The level of detail argument `lod` sets how much of each docstring follows the name: `'terse'` (none, the default), `'short'` (the first four lines), or `'all'`. The short form for the quantile method `q` shows its signature and the first four lines of its docstring, which include the definition of the lower quantile.

```
a.help('^q$', lod='short')
```

```
Callable: q(p, kind='lower')
Return quantile function of density_df.p_total.

Definition 2.1 (Quantiles)
x(α) = qα(X) = inf{x ∈ R : P[X ≤ x] ≥ α} is the lower α-quantile of X
```

The `values` argument adds the value of each matching attribute and property. A method shows its name and signature and is never called, so a broad regex is safe. With `'short'` a DataFrame or Series is cut to five rows, and with `'all'` it is shown in full.

```
a.help('^est_(m|cv)$', values='all')
```

```
Attribute: est_cv
0.28171808490950573
Attribute: est_m
10.5
```

The module-level `build` is an `Underwriter`, and its help method gives a quick index to the recipe base: the methods that load, look up, fork, and preview recipes. The recipe base itself is described in Using recipes.

```
build.help('^(load|recipe|recipes|fork|preview)$')
```

```
Callable: fork(name=None)
Callable: load(request=None)
Callable: preview(program)
Callable: recipe(name, kind='')
Attribute: recipes
```

The `fmt` argument sets the output format. The default, `'auto'`, gives ANSI color under Jupyter and plain text elsewhere; `'text'` and `'ansi'` force one or the other, and `'html'` renders each docstring as Markdown in a notebook. Python's own `help(a.q)` prints the full docstring of a single method.

Parse error reports

A DecL program that does not parse raises a `ValueError`. Its message is a one-line summary, including any suggestion, and its `report` attribute is an `ErrorReport` containing the line and column, the offending token, the suggestions, and the tokens that would have been accepted. The underwriter also logs `report.render()` at level ERROR before raising, so the full display, with a caret under the error, appears on stderr above the output of the cell.

```
try:
    build('agg Typo 10 clams sev lognorm 10 cv 2 poisson')
except ValueError as e:
    report = e.report
    wprint(e)
print(report.line, report.column, report.got, report.suggestions)
wprint(report.expected)
```

```
DecL parse error at line 1, column 13:

  agg Typo 10 clams sev lognorm 10 cv 2 poisson
              ^^^^^

Unexpected 'clams'. Did you mean: claims? Expected: 'claims' (or 'claim'), '/', '**' or '^',
'exposure', '@', 'loss', 'premium' (or 'prem'), '*', 'years' (or 'year'; renewal horizon)

DecL parse error at line 1, column 13: Unexpected 'clams'. Did you mean: claims?
1 13 clams ['claims']
["'claims' (or 'claim')", "'/'", "'**' or '^'", "'exposure'", "'@'", "'loss'",
"'premium' (or 'prem')", "'*'", "'years' (or 'year'; renewal horizon)"]
```

Some parse errors have a token report that is accurate but unhelpful, and for a known trap of this kind the report adds a hint with the rule that was broken. In an expense clause `as` closes an expense group, so an `and` after the label cannot parse, and the hint includes both fixes. The hint is `report.hint`, and it is appended to the summary, so `str(e)` includes it.

```
program = ('xpnl E.X 10000 premium less agg B.A 6500 loss 1000 xs 0 '
           'sev lognorm 100 cv 2 mixed ig 0.25 less '
           '500 fixed expense as FE and 15% premium expense as Comm')
try:
    build(program)
except ValueError as e:
    wprint(e)
```

```
DecL parse error at line 1, column 121:

  ... mixed ig 0.25 less 500 fixed expense as FE and 15% premium expense as Comm
                                                  ^^^

Unexpected 'and'. Expected: 'exp', a hints clause (hints{...}), 'lpar', 'lsqb', a note clause
(note{...}), a number, 'peel' (walk reinsurance layers, not tiers), a tags clause (tags{...})
Hint: 'as' closes an expense group, so 'and' cannot follow the label. Drop the 'and' to leave the
groups separate (one expense leg each), or move the 'as' label after the last 'and'-joined term to
combine them into one leg.

DecL parse error at line 1, column 121: Unexpected 'and'. 'as' closes an expense
group, so 'and' cannot follow the label. Drop the 'and' to leave the groups
separate (one expense leg each), or move the 'as' label after the last
'and'-joined term to combine them into one leg.
```

Two errors the grammar cannot catch have the same report. A misspelled distribution name, such as `lognorml`, is a legal identifier to the grammar, so it is checked against the distribution families, with the closest names in `report.suggestions`. An unmatched bracket is reported at its position in the program.

```
for program in ['agg Typo 10 claims sev lognorml 10 cv 2 poisson',
                'agg Open 1 claim dsev [1 2 fixed']:
    try:
        build(program)
```

```
    except ValueError as e:
        wprint(e)
        print(e.report.line, e.report.column, e.report.got, e.report.suggestions)
```

```
DecL parse error at line 1, column 24:

  agg Typo 10 claims sev lognorml 10 cv 2 poisson
                         ^^^^^^^^

Unknown severity family 'lognorml'. Did you mean: lognorm, loguniform, powerlognorm?

DecL parse error at line 1, column 24: Unknown severity family 'lognorml'. Did
you mean: lognorm, loguniform, powerlognorm?
1 24 lognorml ['lognorm', 'loguniform', 'powerlognorm']
DecL parse error at line 1, column 23: Unmatched '['.
1 23 [ []
```

Refused help requests

An unknown help page, level of detail, or output format is refused. Each raises a `ValueError` that lists the accepted values.

```
refuse(lambda: decl_help('cheat'), lambda: a.help(lod='long'),
       lambda: a.help(fmt='pdf'))
```

```
Unknown DecL help form 'cheat'; the forms are 'anatomy', 'card'.
lod must be 'terse', 'short', or 'all'; got 'long'
fmt must be 'auto', 'text', 'ansi', or 'html'; got 'pdf'
```

Using labels

Display labels on statements, layers, reinsurance, expenses, and premium with the DecL as clause

Stephen J. Mildenhall

2026-10-30

Every DecL statement that defines an object has a name, the bare word after its kind, which identifies the object and is the handle another program uses to refer to it. A **label** is the text shown for it in output. The clause `as LABEL` follows the item it labels: the name of a statement, an occurrence layer, a reinsurance layer, an expense group, or a premium. A label is presentation only and changes no computed value. The `label` property returns the label, or a default, or the name, so it is never blank, and `qd` shows it in the heading of its description.

A label that is a single word, such as `as GWP`, needs no quotes. A label with spaces or punctuation is quoted, `as "Motor Deal"`, and may contain any character except a double quote or a line break; a reserved word such as `premium` must also be quoted. Without a label the output uses a default: `Gross` for the first step of a P&L walk, `Premium` and `Loss` for its premium and loss legs, the basis for an expense group, such as `fixed expense`, and the layer, such as `occ 1750 xs 250`, for a reinsurance step.

```
xpnl Deal as "Motor Deal" 10000 premium as GWP less ...     # statement and premium
agg Deal.Book as "Motor Book" ...                           # the wrapped aggregate
2000 xs 0 as "Policy Limit"                                 # occurrence layer
occurrence net of 1750 xs 250 rate 0.325 as "Occ Re"        # reinsurance layer
0.15 premium expenses as Acquisition                        # expense group
```

```
import re
from aggregate import build, qd, __version__
from style import wprint, GT, figures
figures()
print(__version__)
```

```
1.43.1
```

Labels in a P&L walk

The example is the ceding commission treaty of Using pnl and xpnl, with a label at every site and the expense split into two groups. In the `qd` description the first step is the label of the `xpnl`, `Motor Deal`, and the cover step is the label of the reinsurance layer, `Occ Re`.

```
program = '''xpnl Deal as "Motor Deal"
  10000 premium as GWP
  less
    agg Deal.Book as "Motor Book"
      10000 premium at 0.7 lr
```

```
      2000 xs 0 as "Policy Limit"
      sev lognorm 150 cv 2.5
      occurrence net of
        1750 xs 250 rate 0.325 cede 0.1 as "Occ Re"
      mixed gamma 0.25
  less
    0.15 premium expenses as Acquisition
    1000 fixed expenses as "General Expense"'''
x = build(program)
qd(x)
print(x.name, x.label, sep=', ')
```

```
PnL('Motor Deal': 2 groups, 7 legs -> 'margin')

                             EX     SD       CV     Skew    P01  Median   P99
Step       Side
Motor Deal Consideration  10000      0        0           10000   10000 10000
           Obligation     -9500 2706.5 -0.28489 -0.67617 -17060   -9180 -4620
           Margin           500 2706.5   5.4129 -0.67617  -7060     820  5380
Occ Re     Consideration  -3250      0        0           -3250   -3250 -3250
           Obligation      2844 1619.5  0.56944  0.94221    405    2605  7645
           Margin          -406 1619.5  -3.9889  0.94221  -2845    -645  4395
           Net           93.999 1426.7   15.178  -0.5251  -3745     215  2855
All        Consideration   6750      0        0            6750    6750  6750
           Obligation     -6656 1426.7 -0.21435  -0.5251 -10495   -6535 -3895
           Margin        93.999 1426.7   15.178  -0.5251  -3745     215  2855
           Impact          -406 1619.5  -3.9889  0.94221  -2845    -645  4395
Deal, Motor Deal
```

The ledger `economic_df` shows the labels in its `Label` level. The premium leg is `GWP`, the loss leg and the margin of the first step take the label of the wrapped aggregate, `Motor Book`, each expense group is its own leg, and the legs of the cover are `Occ Re premium`, `Occ Re recovery`, and `Occ Re commission`. The rows `Total`, `Net`, and `Impact` are derived and keep their names. The rows of `economic_ratios_df` are the steps, under the same labels.

```
GT(x.economic_df[['EX', 'SD']], float_format=',.0f')
```

Step	*Side*	*Label*	EX	SD
Motor Deal	Consideration	GWP	10,000	0
	Obligation	Motor Book	-7,000	2,706
		Acquisition	-1,500	0
		General Expense	-1,000	0
		Total	-9,500	2,706
	Margin	Motor Book	500	2,706
Occ Re	Consideration	Occ Re premium	-3,250	0
	Obligation	Occ Re recovery	2,519	1,619
		Occ Re commission	325	0
		Total	2,844	1,619
	Margin	Total	-406	1,619
		Net	94	1,427
All	Consideration	Net	6,750	0
	Obligation	Net	-6,656	1,427
	Margin	Net	94	1,427
		Impact	-406	1,619

```
GT(x.economic_ratios_df[['P', 'L', 'E', 'M', 'LR', 'CR']],
   float_format=',.0f', ratio_cols=['LR', 'CR'])
```

Step	P	L	E	M	LR	CR
Motor Deal	10,000	7,000	2,500	500	70.0%	95.0%
Occ Re	-3,250	-2,519	-325	-406	77.5%	87.5%
All	6,750	4,481	2,175	94	66.4%	98.6%

Default labels and labels on the wrapped aggregate

The same program with every `as` clause removed shows the defaults, first for the steps and then in the `Label` level of the ledger. The labels of the occurrence layer and of the reinsurance layer are stored on the wrapped aggregate, in its `labels`.

```
plain = build(re.sub(r' as ("[^"]*"|\w+)', '', program).replace('Deal', 'Plain'))
print(list(plain.economic_ratios_df.index))
wprint(list(plain.economic_df.index.get_level_values('Label').unique()))
print(x.engine.labels)
```

```
['Gross', 'occ 1750 xs 250', 'All']
['Premium', 'Loss', 'premium expense', 'fixed expense', 'Total', 'Gross', 'occ
1750 xs 250 premium', 'occ 1750 xs 250 recovery', 'occ 1750 xs 250 commission',
'Net', 'Impact']
labels({'layer': 'Policy Limit', 'occ_reins': {0: 'Occ Re'}})
```

Using lattice buckets

Exact bucket sizes for discrete severities on a dyadic lattice

Stephen J. Mildenhall

2026-11-01

A discrete severity whose atoms are all whole multiples of a step h has an aggregate on the same lattice $h\mathbb{Z}$, whatever the frequency. An exact and small grid therefore exists: put every atom on a grid point and size the grid to hold the support and one claim. The bucket sizer[1] (Using FFT windows) finds that grid for integer, fractional, and signed atoms, with one rule for an `Aggregate`, a `Portfolio`, and a discrete bivariate axis.

The rule has three parts. **The step.** Every float is a dyadic rational, so "on a lattice" needs a depth and a tolerance. Let $M = \max|x_j|$. Search the powers of two 2^{-r} from the coarsest at or below M down to $2^{-D}M$, where D is the larger of 16 and the requested `log2`, and take the first at which every atom is within $10^{-12}M$ of $2^{-r}\mathbb{Z}$. The step is $h = \gcd_i(p_i)/2^r$, with p_i the rounded $x_i 2^r$, written $h = n/2^s$ with n odd. Decimal atoms such as 0.1, and thirds, never meet the test and have no lattice. The atoms are those of every view the grid holds: gross after any layer, ceded and net of each occurrence layer, and the points of an aggregate cover. **The candidates.** In order of preference, the candidate buckets are the largest nice bucket dividing h and then h itself. A nice bucket is one in the set used by `round_bucket`: 2^{-s} for a fractional step (0.75 gives 0.25, 1.5 gives 0.5), and the largest of $\{1, 2, 4, 5, 8\} \cdot 10^k$ dividing an integer step (3 gives 1, 15 gives 5, and 1000 stays 1000). **The fit.** The sizer takes the first candidate whose grid holds both the window and one claim within the requested `log2`, then shrinks `log2` to what those two need. If no candidate fits, the standard sizing methods select the bucket, the atoms do not sit on the grid, and the sizer warns.

```
import warnings
import numpy as np
import pandas as pd
import scipy.stats as ss
from aggregate import build, Aggregate, Portfolio, __version__
from aggregate.utilities import _dyadic_lattice_step, _lattice_candidates
from style import wprint, GT, figures, refuse
figures()
print(__version__)
```

```
1.58.0
```

Lattice buckets for three-atom severities

Each example has a three-atom severity with probabilities 1/4, 1/2, and 1/4 and a fixed or Poisson count, mostly of 3 claims. The column `cap` is the requested `log2`, `step` is the lattice step h, and `on_grid` shows whether every atom lands on a grid point.

[1] From aggregate 1.26.0

```
rows = [  # atoms, claim count, frequency, log2 cap
    ('0.25 0.5 1.75', 3, 'fixed', 16), ('-1.75 0.25 0.5', 3, 'fixed', 16),
    ('0.75 1.5 2.25', 3, 'fixed', 16), ('0.75 1.5 2.25', 3, 'fixed', 4),
    ('1.5 3 4.5', 3, 'fixed', 16), ('3 6 9', 3, 'fixed', 16),
    ('3 6 9', 3, 'poisson', 16), ('3 6 9', 1000000, 'poisson', 16),
    ('5 10 15', 3, 'fixed', 16), ('5 10 15', 3, 'poisson', 16),
    ('1000 2000 5000', 3, 'fixed', 16), ('1000 2000 5000', 3, 'poisson', 16),
    ('0.1 0.2 0.3', 3, 'fixed', 16)]
out = []
for atoms, n, freq, cap in rows:
    a = build(f'agg T {n} claims dsev [{atoms}] [.25 .5 .25] {freq}', log2=cap)
    k = (np.array(atoms.split(), dtype=float) - a.x_min) / a.bs
    out.append(dict(atoms=atoms, claims=n, freq=freq, cap=cap,
                    step=a._severity_lattice(), bs=a.bs, log2=a.log2,
                    on_grid=bool(np.array_equal(k, np.round(k)))))
table = pd.DataFrame(out)
GT(table, float_format='.6g')
```

atoms	claims	freq	cap	step	bs	log2	on_grid
0.25 0.5 1.75	3	fixed	16	0.25	0.25	5	True
-1.75 0.25 0.5	3	fixed	16	0.25	0.25	5	True
0.75 1.5 2.25	3	fixed	16	0.75	0.25	5	True
0.75 1.5 2.25	3	fixed	4	0.75	0.75	4	True
1.5 3 4.5	3	fixed	16	1.5	0.5	5	True
3 6 9	3	fixed	16	3	1	5	True
3 6 9	3	poisson	16	3	1	8	True
3 6 9	1,000,000	poisson	16	3	3	15	True
5 10 15	3	fixed	16	5	5	4	True
5 10 15	3	poisson	16	5	5	6	True
1000 2000 5000	3	fixed	16	1000	1000	4	True
1000 2000 5000	3	poisson	16	1000	1000	7	True
0.1 0.2 0.3	3	fixed	16	—	1.52588e-05	16	False

The quarter-point atoms need only 32 buckets of 0.25. The decimal atoms 0.1, 0.2, and 0.3 have no lattice and take the standard bucket 2^{-16} on 65,536 buckets, with the atoms off the grid. A lattice of 0.75 takes the nice bucket 0.25, unless the cap is too tight for it: at `log2` 4 it takes the step 0.75 itself. A fixed and a Poisson count on the same atoms get the same bucket: 5 for the atoms 5, 10, and 15, and 1 for the atoms 3, 6, and 9.

The aggregate with a million claims keeps the step 3 as its bucket. Its count, not its atoms, sets the width of its band of mass, 75,052, and the nice bucket 1 would need `log2` 17, more than the cap of 16. The nice bucket is held to the cap, so `log2` never grows just to make the bucket nicer, and the step 3 fits on 2^{15} buckets. Only the exact step may take `log2` past the cap, and only in the `windowed` sizing method.

```
a = build('agg T 1000000 claims dsev [3 6 9] [.25 .5 .25] poisson')
w = a.bs_window_df.loc['windowed']
print(a.bs, a.log2, f'band width {w["W"]:,.0f}',
      'log2 needed at bs 1:', int(np.ceil(np.log2(w['W'] + 1))))
```

```
3.0 15 band width 75,052 log2 needed at bs 1: 17
```

A windowed grid that holds one claim

A concentrated aggregate has all its mass in a narrow band far above 0, and the `windowed` sizing method (Using FFT windows) places the grid over that band. The severity is discretized on the same number of buckets wherever the grid starts, so the grid must span one claim as well as the band. One hundred claims on the atoms 1000 and 1001 have a band about 59 wide, which needs `log2` 6 at bucket 1, while a claim of 1001 needs `log2` 10. The sizer takes the larger of the two. The aggregate is then the exact binomial law on 100,000 to 100,100, on a grid starting at 99,640.

```
a = build('agg T 100 claims dsev [1000 1001] fixed')
wprint(a.bs_window_df.loc['windowed', 'note'])
xs = np.arange(100000, 100101)
err = a.density_df.p_total.reindex(xs) - ss.binom(100, 0.5).pmf(xs - 100000)
print(a.bs, a.log2, a.x_min, f'largest error {err.abs().max():.1e}')
```

```
two-sided window, benign FFT wrap; lattice 1 (residual 0), bs 1 on it;
sev_hull=1001, extent=1024
1.0 10 99640.0 largest error 7.1e-16
```

The lattice step and candidate buckets

The private functions `_dyadic_lattice_step` and `_lattice_candidates` in `aggregate.utilities` implement the first two parts of the rule. The first returns the step and the largest distance of an atom from its lattice point, or `None` when there is no lattice. The second returns the candidate buckets in order of preference.

```
for atoms in ([0.25, 0.5, 1.75], [0.75, 1.5, 2.25], [3, 6, 9],
              [200000, 400000, 1000000], [0.1, 0.2, 0.3], [1 / 3, 2 / 3],
              [0, 1, 100000]):
    fit = _dyadic_lattice_step(atoms)
    print(f'{str(atoms):32s} {str(fit):24s}',
          _lattice_candidates(fit[0]) if fit else '')
```

```
[0.25, 0.5, 1.75]                (0.25, 0.0)              [0.25]
[0.75, 1.5, 2.25]                (0.75, 0.0)              [0.25, 0.75]
[3, 6, 9]                        (3.0, 0.0)               [1.0, 3.0]
[200000, 400000, 1000000]        (200000.0, 0.0)          [200000.0]
[0.1, 0.2, 0.3]                  None
[0.3333333333333333, 0.6666666666666666] None
[0, 1, 100000]                   None
```

The search starts at the largest atom, so the test does not depend on scale: atoms in the hundreds of thousands find their step just as small integers do. The atoms 0, 1 and 100,000 have the integer step 1, but it is more than 16 binary levels below the largest atom, too fine to be a sensible bucket at the default depth, so the search finds no lattice.

Lattice notes in bs_window_df

In the frame `bs_window_df` each sizing method that used the lattice records it in the `note` column, with the step, the residual, and whether the bucket divides the step. The string `bs_explanation` gives the winning method and the window.

```
a = build('agg T 3 claims dsev [0.75 1.5 2.25] [.25 .5 .25] fixed')
for method, row in a.bs_window_df.iterrows():
    wprint(f'{method:15s} {row["bs"]:<8g} {row["note"]}')
wprint(a.bs_explanation)
```

```
moment          0.25     3-moment MoM window; lattice 0.75 (residual 0), bs 1/4
                on it
exact_discrete  0.25     dfreq/fixed x dsev on a lattice; lattice 0.75 (residual
                0), bs 1/4 on it; logp_lo=-1.806, logp_hi=-1.806
bounded_small   0.25     bounded severity x freq quantile; lattice 0.75
                (residual 0), bs 1/4 on it
sbj             0.25     single big jump: ES - mu_X + q_X(p**)
used            0.25     realized grid (exact_discrete)
The aggregate is severity-driven, [2.25, 6.75], bounded. Log2 is 5. Of the
methods that applied (moment 10.9616, exact_discrete 6.75, bounded_small 6.75,
sbj 8.31102), the exact_discrete method won (window width 6.75) -- the exact
discrete (lattice) support. It has a natural lower support bound of 2.25. It has
a natural upper support bound of 6.75. The recommended x_min is 0 and the x_max
that follows is 8, so the window is [0, 8] of width 8.
```

A bucket given explicitly, here `bs=0.5`, is used as given, and the grid keeps the requested `log2` of 16. Setting `bs` is the way to override the lattice. If the bucket misses the lattice, the note records the miss. No lattice warning is raised, because the bucket was chosen, and the wrong distribution shows in the verdict and its warning (Using validation).

```
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter('always')
    a = build('agg P 3 claims dsev [0.75 1.5 2.25] [.25 .5 .25] fixed', bs=0.5)
print(a.bs, a.log2)
wprint(a.bs_window_df.loc['exact_discrete', 'note'])
wprint(str(caught[0].message))
```

```
0.5 16
dfreq/fixed x dsev on a lattice; pinned bs 1/2 misses the lattice 0.75;
logp_lo=-1.806, logp_hi=-1.806
P: fails sev cv, agg cv. See validation_explanation. Further occurrences this
session are suppressed; each object records its own verdict in validation.
```

Atoms with floating-point noise

Arithmetic that should land on the lattice, such as a conditional mean or a scaled atom, often misses it by a few units in the last place. The tolerance $10^{-12}M$ treats the error as noise, not as a finer lattice: the bucket is the clean one, the residual goes in the note, and the mass of each atom is in one bucket under both `dsev_bucket` schemes.

```
noisy = '0.25000000000000006 0.5000000000000001 1.7499999999999998'
for scheme in ('linear', 'nearest'):
    a = build(f'agg N 1 claim dsev [{noisy}] [.25 .5 .25] fixed',
              dsev_bucket=scheme)
    print(scheme, a.bs, a.log2, 'nonzero buckets', (a.sev_density > 0).sum())
wprint(a.bs_window_df.loc['exact_discrete', 'note'])
```

```
linear 0.25 3 nonzero buckets 3
nearest 0.25 3 nonzero buckets 3
dfreq/fixed x dsev on a lattice; lattice 1/4 (residual 2.2e-16), bs 1/4 on it;
logp_lo=-0.6021, logp_hi=-0.6021
```

Reinsured discrete severities

A layer or a reinsurance cover changes the amounts the grid must hold, so the lattice step is taken over every view of the severity. The atoms 1000, 2000, and 3000 net of an occurrence layer `1500 xs 250` are 250, 500, and 1500, and the ceded amounts are 750 and 1500. The step of all these amounts is 250, whose largest nice divisor is 50. A share that leaves the dyadic lattice, such as `30% po` of integer atoms, has no step, and the standard sizing methods select the bucket.

```
a = build('agg N 1 claim dsev [1000 2000 3000] occurrence net of 1500 xs 250 fixed')
p = a.density_df.p_total
print(a._severity_lattice(), a.bs, a.log2, p[p > 1e-12].round(4).to_dict())
```

```
250.0 50.0 6 {250.0: 0.3333, 500.0: 0.3333, 1500.0: 0.3333}
```

A reinsured aggregate is sized from 0 and does not use a window, because a ceded amount can fall below the window. The next aggregate has a gross mean of 27 and cedes 2 on every outcome.

```
a = build('agg A 3 claims dsev [8 10] fixed aggregate ceded to 2 xs 4')
p = a.density_df.p_total
print(a.bs, a.log2, a.x_min, p[p > 1e-12].round(4).to_dict())
```

```
2.0 4 0.0 {2.0: 1.0}
```

A lattice too fine for the log2 cap

An exact grid exists for every lattice, but it may not fit the requested `log2`. The aggregate below has support up to 100,100, which needs `log2` 17 at the step 1. At the default cap of 16 it builds on bucket 2, with the atom at 1001 between grid points. The build completes, because an aggregate is often one unit of something larger. The warning `LatticeGridWarning`[2] gives the `log2` to pass, and the verdict contains the same numbers. The kind `nttagg` (Using `nttagg`) raises an error in the same position.

```
program = 'agg AWK dfreq [1:100] dsev [1000 1001]'
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter('always')
    a = build(program)
print(a.bs, a.log2)
wprint(str(caught[0].message))
wprint(a.validation_description)
a = build(program, log2=17)
print(a.bs, a.log2, a.validation_description)
```

```
2.0 16
AWK: the severity atoms are on the lattice of step 1 and the grid needs log2 >=
17 at bs 1; log2 16 was given, so bs 2 is used and the atoms do not sit on the
grid. Pass build(..., log2=17) or add hints{log2=17} to the declaration. Further
occurrences this session are suppressed; each object records its own verdict in
validation.
grid infeasible for this lattice (needs log2 17 at bs 1); fails sev cv, agg skew
1.0 17 not unreasonable
```

[2] From aggregate 1.48.0

Lattice buckets for a Portfolio

A `Portfolio` whose units are all discrete takes the step of their pooled atoms, and the rule selects the bucket once, against the window of the `Portfolio`. Its grid must hold the largest claim of any unit, it does not use a window when a unit is reinsured, and it warns as an `Aggregate` does when the lattice does not fit. Poisson units on steps 2 and 5 pool to step 1, so every atom of both units is a whole number of buckets from the origin.

```
p = build('port Steps agg S2 3 claims dsev [2 4 6] poisson '
          'agg S5 3 claims dsev [5 10 15] poisson')
x0 = float(p.density_df.index[0])
print(p.bs, p.log2, (np.array([2, 4, 6, 5, 10, 15]) - x0) / p.bs)
```

```
1.0 9 [ 2.  4.  6.  5. 10. 15.]
```

A `Portfolio` of one unit takes the grid of that unit, so the two objects hold the same law on the same grid.

```
a = build('agg A dfreq [1] dsev [3]')
p = build('port P agg A dfreq [1] dsev [3]')
print(a.bs, a.log2, a.x_min, '|', p.bs, p.log2, float(p.density_df.index[0]))
```

```
1.0 2 0.0 | 1.0 2 0.0
```

A signed `Portfolio` of units on quarter points gets the bucket 0.25, and `log2` shrinks to the 8 it needs. Every atom of every unit is again a whole number of buckets from the origin of the grid, −32.75.

```
p = build('port Signed '
          'agg A 3 claims dsev [-1.75 0.25 0.5] [.25 .5 .25] poisson '
          'agg B 2 claims dsev [-0.5 0.75] poisson '
          'agg C 1 claim dsev [0.25 1.5] fixed')
x0 = float(p.density_df.index[0])
atoms = np.array([-1.75, 0.25, 0.5, -0.5, 0.75, 1.5])
print(p.bs, p.log2, x0, (atoms - x0) / p.bs)
wprint(p.bs_explanation)
```

```
0.25 8 -32.75 [124. 132. 133. 129. 134. 137.]
The portfolio tail is super-exponential left / super-exponential right. Log2 is
8. The unit tails are: A super-exponential/super-exponential and B
super-exponential/super-exponential and C bounded/bounded. The recommended
window width 64 is based on portfolio method of moments 32.9446, RMS(units)
37.4895, sum(units) 50.8032, and single big jump of 8.08653. Every unit is
discrete and the pooled atoms sit on a lattice of 1/4, so bs is 1/4, the first
of its lattice buckets whose grid fits log2 8, and every atom lands on a grid
point. The recommended x_min is -24.25 and the x_max that follows is 39.75, so
the window is [-24.25, 39.75] of width 64.
```

A Portfolio of many signed units

The grid of a signed `Portfolio` starts at the sum of the origins of its units, which with many units is far below the moment window. Two hundred units on the atoms −1.75, 0.25, and 0.5 start at −350, while the window starts near −136. A `log2` measured from the window would be 10, putting the grid on $[-350, -94)$, below most of the mass. The sizer measures the need from the realized origin instead and takes `log2` 11, so the `Portfolio` keeps all of its mass and its exact mean, −37.5. A `Portfolio` of many signed units with no lattice can still lose mass in this way.

```
aggs = [Aggregate(name=f'U{i}', exp_en=1, freq_name='fixed',
                  sev_name='dhistogram', sev_xs=[-1.75, 0.25, 0.5],
                  sev_ps=[.25, .5, .25]) for i in range(200)]
p = Portfolio('Many', aggs)
p.update(log2=16, bs=0)
s = p.density_df.p_total
print(p.bs, p.log2, float(p.density_df.index[0]))
print(f'mass {s.sum():.15f}, mean {(s * s.index).sum():.10f}, exact {-0.1875 * 200}')
```

```
p_total has 0 negative values; NOT setting to zero...

0.25 11 -350.0
mass 1.000000000000006, mean -37.5000000000, exact -37.5
```

Bivariate axes on and off a lattice

A discrete bivariate axis on a lattice takes the bucket of the rule, and the given joint matrix lands at exact indices. An axis off any lattice is sized freely, like a copula axis, and the matrix is spread over it with the units' `dsev_bucket` scheme. The margins of the bivariate severity, the row and column sums of the matrix `S` below, still equal the marginal severities `g0` and `g1`. The axes `[0 0.1 0.2]` and `[0 0.1 0.3000001]` have no lattice and get 1,024 buckets of 2^{-11} and 2^{-10}.

```
for axis in ['[0 1 2]', '[0.25 0.75]', '[0 0.1 0.2]', '[0 0.1 0.3000001]']:
    b = build(f'bivariate BV dfreq [1 2] [.5 .5] dbvsev {axis} [0 1 2]')
    S = np.asarray(b._S)
    g0, g1 = b._gs
    margins = (np.allclose(S.sum(axis=1), g0, rtol=0, atol=1e-14)
               and np.allclose(S.sum(axis=0), g1, rtol=0, atol=1e-14))
    wprint(f'{axis:20s} bs {b.bs}, on lattice {b._on_lattice}, '
           f'grid {[len(x) for x in b.axis_xs]}, margins match {margins}')
```

```
[0 1 2]              bs [1.0, 1.0], on lattice [True, True], grid [16, 16],
                     margins match True
[0.25 0.75]          bs [0.25, 1.0], on lattice [True, True], grid [16, 16],
                     margins match True
[0 0.1 0.2]          bs [0.00048828125, 1.0], on lattice [False, True], grid
                     [1024, 16], margins match True
[0 0.1 0.3000001]    bs [0.0009765625, 1.0], on lattice [False, True], grid
                     [1024, 16], margins match True
```

Probabilities of a discrete severity that do not sum to 1

The probabilities of a `dsev` clause must sum to 1. A sum within a relative 10^{-5} of 1 is normalized when the severity is constructed, so probabilities printed to six digits give the moments of the law they round. The first line below shows the standard deviation and skewness of 100 claims on the atoms 999, 1000, and 1001 with such probabilities. A sum outside that tolerance is refused.

```
a = build('agg R 100 claims dsev [999 1000 1001] [0.333333 0.222222 0.444444] fixed')
print(f'{a.actual_sd:.4f} {a.actual_skew:.4f}')
refuse('agg Q 2 claims dsev [1 2] [0.3 0.3] fixed')
```

```
8.7489 -0.0213
dsev probabilities sum to 0.6; they must sum to 1, to within a relative 1e-5.
```

Using loss corridors

Loss ratio corridors retained by the cedant on an aggregate quota share with the DecL corridor clause

Stephen J. Mildenhall

2026-10-30

A **loss corridor** is a band of loss ratio in which the cedant takes back part of the ceded loss. The reinsurer pays in full below the band and only in part inside it, and above the band it pays in full again, short by the amount the cedant retained. Corridors appear on quota shares, often of new or volatile business, and on some excess of loss treaties, usually with the band at or above the expected loss ratio. With ceded loss A, a fixed ceded premium P_C and ceded loss ratio $\lambda = A/P_C$, the cedant retains a share s of a band of width w above an attachment a, and the ceded loss ratio after the corridor is

$$\lambda' = \lambda - s\min((\lambda - a)^+, w),$$

with recovery $A' = \lambda' P_C$. The retained amount is s times a layer of the loss ratio, w excess of a, and is at most swP_C.

In DecL the clause `corridor <share> po <width> xs <attach>` decorates one layer of an `aggregate net of` program, with the band written in loss ratio terms and the layer words `po` and `xs`. The layer needs a `deposit`, `rol`, or `rate` premium to form the loss ratio, and it may take a flat commission `cede` beside the corridor. The corridor is one of the variable features of DecL, with sliding scale commissions, profit commissions, swing rating, and retro rating. A program takes at most one of them, the reinsurance features decorate aggregate covers only, and all take effect in a `pnl` or `xpnl` (see Using pnl and xpnl). The corridor changes the recovery, booked in the loss column `L` of `economic_ratios_df`.

```
aggregate net of 50% po inf xs 0 rate 100% cede 0.25 corridor 0.5 po 0.2 xs 0.6 as QS
aggregate net of 50% po inf xs 0 rate 100% cede 0.25 corridor 100% po 10% xs 75% as QS
aggregate net of 2000 xs 8000 rate 6% corridor 0.5 po 0.5 xs 1 as "Stop loss"
```

The variable feature pages share one base book: premium 10,000 at a 70% loss ratio, a policy limit of 2,000, lognormal severity with mean 150 and CV 2.5, gamma mixed claim counts with mixing CV 0.25, and expenses of 25% of premium. The setup cell wraps the book in an `xpnl` with a slot for each reinsurance program.

```
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
book = '''agg Book
  10000 premium at 0.7 lr
  2000 xs 0
  sev lognorm 150 cv 2.5 {occurrence}
```

```
  mixed gamma 0.25 {aggregate}'''
program = '''xpnl {name}
  inherit premium
  less
    ''' + book + '''
  less
    0.25 premium expenses'''
def quota_share(name, terms):
    """The base book under a 50% quota share at a 100% rate with the given terms."""
    return build(program.format(name=name, occurrence='',
                 aggregate=f'aggregate net of 50% po inf xs 0 rate 100% {terms} as QS'))
```

```
1.43.1
```

A loss corridor on a 50% quota share

The cover is a 50% quota share of the base book at 100% of the subject premium with a flat 25% commission, so the ceded premium is 5,000 and the ceded loss ratio equals the gross loss ratio, with mean 70%. The cedant retains half the ceded loss in the band of loss ratio from 60% to 80%, at most $0.5 \times 0.2 \times 5{,}000 = 500$. The comparison is the same quota share and commission with no corridor.

```
plain = quota_share('Plain', 'cede 0.25')
corridor = quota_share('Corridor', 'cede 0.25 corridor 0.5 po 0.2 xs 0.6')
print(corridor.engine.variable_terms)
qd(corridor)
```

```
CorridorTerms(share=0.5, width=0.2, attachment=0.6)
PnL('Corridor': 2 groups, 6 legs -> 'margin')

                         EX     SD       CV     Skew     P01  Median     P99
Step  Side
Gross Consideration   10000      0        0            10000   10000   10000
      Obligation      -9500   2704 -0.28463 -0.67687  -17071   -9193   -4633
      Margin            500   2704    5.408 -0.67687   -7071     807    5367
QS    Consideration   -5000      0        0            -5000   -5000   -5000
      Obligation     4521.2 1163.8   0.2574  0.85211  2316.5  4423.2  8035.5
      Margin         -478.8 1163.8  -2.4306  0.85211 -2683.5 -576.75  3035.5
      Net              21.2 1549.7     73.1 -0.55615 -4035.5  230.25  2683.5
All   Consideration    5000      0        0             5000    5000    5000
      Obligation    -4978.8 1549.7 -0.31126 -0.55615 -9035.5 -4769.8 -2316.5
      Margin           21.2 1549.7     73.1 -0.55615 -4035.5  230.25  2683.5
      Impact         -478.8 1163.8  -2.4306  0.85211 -2683.5 -576.75  3035.5
```

The `QS` step is the cover from the cedant's side: the ceded premium of 5,000 against the recovery and the commission of 1,250, which together make the obligation row, with mean 4,521. The ledger `economic_df` lists the recovery after the corridor, with mean 3,271 against the 3,500 of the plain quota share, so the cedant retains 229 on average. In the 1 in 100 bad year, column `κ01`, the cedant retains the whole 500 and the recovery is 6,786.

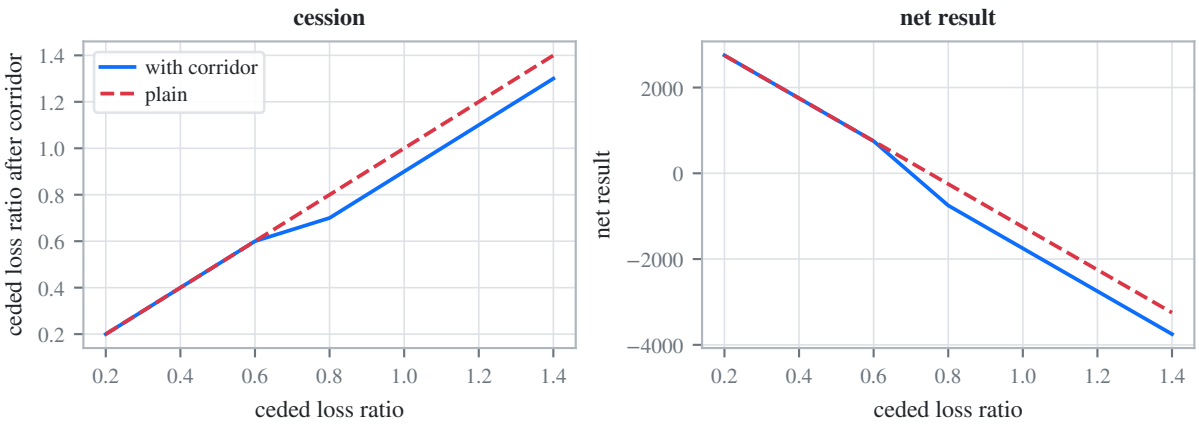
```
GT(corridor.economic_df.loc['QS', ['EX', 'SD', 'κ01', 'κ50', 'κ99']], float_format=',.0f')
```

Side	*Label*	EX	SD	κ01	κ50	κ99
Consideration	QS premium	-5,000	0	-5,000	-5,000	-5,000
Obligation	QS recovery	3,271	1,164	6,786	3,173	1,066
	QS commission	1,250	0	1,250	1,250	1,250
	Total	4,521	1,164	8,036	4,423	2,316
Margin	Total	-479	1,164	3,036	-577	-2,684
	Net	21	1,550	-4,036	230	2,684

Ceded loss ratio and net result against the plain quota share

The ceded loss ratio after the corridor and the cedant's net result are both functions of the ceded loss ratio λ. The gross loss is $10{,}000\lambda$, so the net result, premium less expense and ceded premium, plus commission and recovery, less gross loss, is $2{,}500 + 5{,}000(0.25 + \lambda') - 10{,}000\lambda$. The left panel shows the ceded loss ratio with and without the corridor and the right panel the two net results.

```
lr = np.linspace(0.2, 1.4, 241)
ceded = {'with corridor': corridor.engine.variable_terms.phi(lr), 'plain': lr}
fig, (ax0, ax1) = figure('full', cols=2)
for (k, c), ls in zip(ceded.items(), ['-', '--']):
    ax0.plot(lr, c, ls=ls, label=k)
    ax1.plot(lr, 10000 - 2500 - 5000 + 5000 * (0.25 + c) - 10000 * lr, ls=ls, label=k)
ax0.set(xlabel='ceded loss ratio', ylabel='ceded loss ratio after corridor', title='cession')
ax1.set(xlabel='ceded loss ratio', ylabel='net result', title='net result')
ax0.legend();
```



Inside the band the recovery rises half as fast as the loss, so the net result falls at 1.5 times the slope of the plain quota share. Above 80% the two lines are parallel again, the corridor 500 below. The table compares the two covers over the whole distribution, with `P01` the 1 in 100 bad result.

```
df = pd.DataFrame({k: [-x.economic_ratios_df.loc['QS', 'L'], x.est_m, x.est_sd, x.q(0.01),
x.cdf(0)]
                   for k, x in {'plain': plain, 'with corridor': corridor}.items()},
                  index=['recovery', 'mean', 'SD', 'P01', 'Pr(loss)']).T
df.index.name = 'cover'
GT(df, float_format=',.0f', ratio_cols=['Pr(loss)'])
```

cover	recovery	mean	SD	P01	Pr(loss)
plain	3,500	250	1,352	-3,536	38.5%
with corridor	3,271	21	1,550	-4,036	45.5%

The corridor costs the cedant 229 of mean result, the expected retained loss, and the full 500 in the 1 in 100 year, where `P01` falls from −3,536 to −4,036. The standard deviation (SD) of the net result rises from 1,352 to 1,550 and the probability of a loss, a result of zero or less, from 38.5% to 45.5%.

A full corridor and a sliding scale

A corridor with share 1 over the band from 60% to 80%, beside a flat 35% commission, gives the cedant the same net result as the sliding scale on Using sliding scale commissions, which pays 35% at a loss ratio of 60% and 15% at 80%. Inside the band both take back one point per point of loss ratio, the corridor through the recovery and the sliding scale through the commission, and outside it the two pay the same total. The ledgers differ, but the net results have the same mean and standard deviation.

```
same = quota_share('Same', 'cede 0.35 corridor 1 po 0.2 xs 0.6')
slide = quota_share('Slide', 'slide 0.35 at 0.6 and 0.25 at 0.7 and 0.15 at 0.8')
for x in [same, slide]:
    print(f'{x.name:6s} mean {x.est_m:.4f} SD {x.est_sd:.4f}')
```

```
Same   mean 292.4001 SD 1753.7073
Slide  mean 292.4001 SD 1753.7073
```

Using `nttagg`

Exact compound distributions with the nttagg kind and the number-theoretic transform

Stephen J. Mildenhall

2026-10-30

When the claim count N and the severity X take finitely many nonnegative integer values with rational probabilities, every probability of $S = X_1 + \cdots + X_N$ is rational. Write $\Pr(N = k) = a_k/d_N$ and $\Pr(X = j) = b_j/d_X$, with n the largest count. Then $\mathrm{P}z^S = \sum_k (a_k/d_N)\left(\sum_j b_j z^j/d_X\right)^k$ is a polynomial and $D = d_N d_X^n$ clears its denominators, so $\Pr(S = s) = H_s/D$ with integer H_s. The number-theoretic transform (NTT) is the FFT over $\mathbb{Z}/p\mathbb{Z}$ for primes p with $2^r \mid p-1$, where 2^r is the transform length. It computes each $H_s \bmod p$ exactly, and the Chinese remainder theorem recovers H_s once the product of the primes exceeds D.

The float FFT computes the same polynomial in double precision. Its error is roughly constant in absolute terms, near 10^{-17}, so the relative error in a probability of size 10^{-k} is of order 10^{k-17}: about fifteen figures in the body of the distribution and none below 10^{-17}. A comparison with the exact answer shows where the float answer loses accuracy.

The DecL kind `nttagg`[1] builds an `aggregate.ntt.AggregateNTT`, an ordinary `Aggregate` on the integer lattice, with bucket size 1, that also stores the exact answer. The program has a `dfreq` clause followed by a `dsev` clause, with no exposure, layers, or reinsurance. Outcomes are nonnegative integers, and probabilities are exact as written: `0.3` is 3/10, not the double nearest it. A probability may be an arithmetic expression, such as `(1-1/1000)`, evaluated exactly, and an omitted probability list gives equal exact weights.

```
nttagg Dice dfreq [1:6] dsev [1:6]
nttagg Book dfreq [0 1 2 3] [0.4 0.3 0.2 0.1] dsev [1 2 5] [1/2 1/3 1/6]
nttagg Cat dfreq [0:8] dsev [1 10] [(1-1/1000) 1/1000]
```

```
from fractions import Fraction
from aggregate import build, qd, __version__
from aggregate.ntt import engine_status
from aggregate.charts import build_chart_doc
from aggregate.plots import plot_chartdoc
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.1.0

Exact inputs and exact probabilities

The `Book` example has 0 to 3 claims, each of size 1, 2, or 5. Its `pprogram` shows the program with the probabilities reduced to fractions, and `qd` prints the usual summary.

```
a = build('nttagg Book dfreq [0 1 2 3] [0.4 0.3 0.2 0.1] '
          'dsev [1 2 5] [1/2 1/3 1/6]')
print(a.pprogram)
qd(a)
```

```
nttagg Book
  dfreq [0 1 2 3] [2/5 3/10 1/5 1/10]
  dsev [1 2 5] [1/2 1/3 1/6]
Aggregate object: Book. Frequency distribution empirical. Severity
dhistogram, [1, 5], bounded; atoms [1 2 5]. Updated with bucket size 1
and log2 = 4. Validation: not unreasonable.

      Mean     SD      CV   Skew P01 Median P99
X
Freq     1      1       1    0.6
Sev      2 1.4142 0.70711 1.4142   1      1   5
Agg      2 2.4495  1.2247 1.4153   0      1  10
```

The attribute `p_exact` is a Series of reduced `Fractions` indexed by loss, here over the whole support 0 to 15. Losses of 13 and 14 are impossible, and the largest loss, 15, needs three claims of 5, with probability $(1/10)(1/6)^3 = 1/2160$.

```
wprint(' '.join(str(p) for p in a.p_exact))
```

```
2/5 3/20 3/20 19/240 17/360 1/15 1/27 5/144 1/60 1/180 1/180 1/240 1/360 0 0
1/2160
```

Any quantity computed from the exact probabilities in rational arithmetic is also exact. For `Book` the total probability, the mean, $\Pr(S \le 2)$, and the stop-loss premium $\mathsf{P}(S-5)^+$ are 1, 2, 7/10, and 23/90.

```
pe = {int(x): p for x, p in a.p_exact.items()}
print(sum(pe.values()), sum(x * p for x, p in pe.items()),
      sum(p for x, p in pe.items() if x <= 2),
      sum(max(x - 5, 0) * p for x, p in pe.items()))
```

```
1 2 7/10 23/90
```

The exact moments are a column of `stats_df`, computed in rational arithmetic and rounded to float at the end. Validation compares the FFT moments, in the `empirical` column, with the exact moments. For `Book` the two agree to the six figures shown.

```
GT(a.stats_df.loc['agg', ['exact', 'empirical']], float_format='.6g')
```

measure	exact	empirical
ex1	2	2
ex2	10	10
ex3	64.8	64.8
mean	2	2
cv	1.22474	1.22474
skew	1.41526	1.41526

Exact arithmetic in a probability list

A probability in an `nttagg` list may be an arithmetic expression, and it is evaluated exactly. In the `Cat` example the expression `(1-1/1000)` is exactly 999/1000: a claim is 1 with that probability and 10 otherwise, and there are 0 to 8 claims, each count with probability 1/9. A loss of $10k$ needs exactly k claims, all large, so $\Pr(S = 10k) = 1/(9 \cdot 1000^k)$. A loss of 9 is impossible, and a loss of 8 needs eight small claims.

```
c = build('nttagg Cat dfreq [0:8] dsev [1 10] [(1-1/1000) 1/1000]')
print(c.pprogram)
print(c.p_exact_at([9, 10, 20, 80]))
print(c.p_exact_at(8), c.p_exact_at(8) == Fraction(999, 1000) ** 8 / 9)
print(all(c.p_exact_at(10 * k) == Fraction(1, 9 * 1000 ** k)
          for k in range(1, 9)))
```

```
nttagg Cat
  dfreq [0 1 2 3 4 5 6 7 8]
  dsev [1 10] [999/1000 1/1000]
loss
9.0                                0
10.0                            1/9000
20.0                         1/9000000
80.0    1/900000000000000000000000
Name: p_exact, dtype: object
1102253271188826697768 89/1000000000000000000000000 True
True
```

Accuracy of the float FFT in the tail

For the `Cat` example, the frame `accuracy_df` lists the exact probability, correctly rounded to a float, beside the raw FFT answer, with the relative error computed exactly. The FFT error is never more than about 2×10^{-17}, so its relative error grows as the probability falls, reaching about 9% at 10^{-16}. At smaller probabilities the FFT answer is noise, and it is negative at a loss of 60.

```
df = c.accuracy_df.loc[10:80:10, ['p_exact_float', 'p_fft', 'rel_err']]
GT(df.rename(index=int), float_format='.3e')
```

loss	p_exact_float	p_fft	rel_err
10	1.111e-04	1.111e-04	-3.98e-14
20	1.111e-07	1.111e-07	7.77e-11
30	1.111e-10	1.111e-10	3.22e-08
40	1.111e-13	1.111e-13	-2.89e-05
50	1.111e-16	1.209e-16	8.77e-02
60	1.111e-19	-1.298e-18	-1.27e+01
70	1.111e-22	1.735e-18	1.56e+04
80	1.111e-25	1.605e-17	1.44e+08

The attribute `ntt_audit` summarizes the comparison. The noise floor is the largest absolute FFT value on a bucket whose exact probability is zero, and the maximum relative error is taken over the buckets whose exact probability is above it. The unresolved buckets have a positive exact probability at or below the floor.

```
for k in ['noise_floor', 'max_rel_err', 'rel_err_buckets',
          'unresolved_buckets']:
    print(f'{k:20s} {c.ntt_audit[k]:.4g}')
```

```
noise_floor          2.286e-17
max_rel_err          0.08771
rel_err_buckets      39
unresolved_buckets   6
```

Probabilities below the smallest double

The `Deep` example has the severity of `Cat`, 1 or 10, and a claim count equally likely to be any of 0 to 120. Its largest loss has probability $\Pr(S = 1200) = 1/(121 \cdot 10^{360})$, far below the smallest double, about 5×10^{-324}. The column `p_exact_float` underflows to zero at losses of 1,190 and 1,200, while `p_exact_log10`, computed from the integers, stays finite. The float answer `p_total` is zero at all four losses.

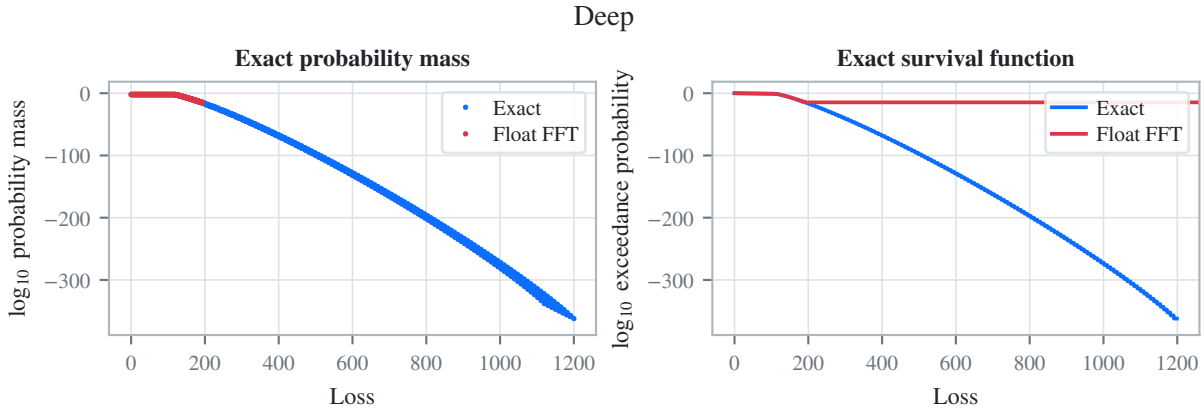
```
d = build('nttagg Deep dfreq [0:120] dsev [1 10] [(1-1/1000) 1/1000]')
print(d.p_exact_at(1200) == Fraction(1, 121 * 10 ** 360))
GT(d.density_df.loc[[300, 600, 1190, 1200],
                    ['p_total', 'p_exact_float', 'p_exact_log10']],
   float_format='.4g')
```

```
True
```

loss	p_total	p_exact_float	p_exact_log10
300	0	2.205e-40	-39.66
600	0	1.031e-131	-131
1,190	0	0	-359.1
1,200	0	0	-362.1

The `exact_tail` chart shows the exact mass and survival function on a base-10 logarithmic scale beside the float FFT answer. The float answer stops near 10^{-15}, and the exact answer continues to 10^{-362}.

```
fig = plot_chartdoc(build_chart_doc(d, 'exact_tail'))
```



The optional compiled package `aggregate-ntt`

The optional compiled package `aggregate-ntt` speeds up the exact calculation in large cases, and the library uses it when it is installed. The function `engine_status()` returns a dictionary showing whether it is used and, if not, why. Passing `ntt_engine='python'` to `build` forces the pure-Python calculation, and for `Deep` the two give identical exact probabilities.

```
wprint(engine_status())
dp = build('nttagg DeepPython dfreq [0:120] dsev [1 10] [(1-1/1000) 1/1000]',
           ntt_engine='python')
print(d.ntt_audit['engine'], dp.ntt_audit['engine'],
      dp.p_exact.equals(d.p_exact))
```

```
{'engine': 'rust', 'installed': True, 'engine_version': '1.0.0',
'engine_protocol': 2, 'library_protocol': 2, 'reason': None}
rust python True
```

Refused programs

An `nttagg` program is refused when its probabilities do not sum to exactly 1, when an outcome is not a nonnegative integer, or when a probability raises a number to a fractional power, whose result need not be rational. The support must also fit the grid, which `build` limits to 2^{16} buckets by default. An `nttagg` cannot be a unit of a portfolio.

```
refuse('nttagg X dfreq [0 1] [1/2 1/3] dsev [1]',
       'nttagg X dfreq [0 1] dsev [1.5 2]',
       'nttagg X dfreq [0 1] dsev [1 2] [(1/2)**(1/2) 1/2]',
       'nttagg X dfreq [1:100] dsev [1:100000]',
       'port P nttagg X dfreq [1] dsev [1]')
```

```
nttagg frequency: probabilities must sum to exactly 1; shortfall is 1/6
nttagg outcomes must be nonnegative integers, got '1.5'; clear denominators and
rescale the lattice yourself
nttagg exponent must be a whole number, got 1/2
X: the exact support 0..10000000 needs log2 >= 24, got 16. Pass build(...,
log2=24) or add hints{log2=24} to the declaration.
DecL parse error at line 1, column 8: Unexpected 'nttagg'. Did you mean: agg,
tags?
```

Using occurrence reinsurance

Per-occurrence excess of loss layers, partial placements and towers (occurrence net of, occurrence ceded to)

Stephen J. Mildenhall

2026-11-01

An occurrence layer `y xs a` placed at share s pays $s\min((x-a)^+, y)$ on a claim of size x. A program of non-overlapping layers pays the sum, $c(x) = \sum_j s_j \min((x-a_j)^+, y_j)$, and the cedant keeps $n(x) = x - c(x)$. Over a year with claims $X_1, \dots, X_N$ the ceded loss is $C = \sum_i c(X_i)$ and the net loss is $\sum_i n(X_i)$, and the two add to the gross loss. Each view is an ordinary compound distribution, with the same claim count and the severity transformed by c or n, so `aggregate` computes it by the same convolution as the gross.

The clause `occurrence net of` makes the aggregate the net view and `occurrence ceded to` makes it the ceded view. The clause follows the severity and precedes the frequency, and every layer is written bottom up. A partial placement is written with `po` (part of): a percentage is the share, and a bare number is an amount, so `250 po 500 xs 500` is a 50% share. Layers are joined by `and`, and `tower` followed by a list of breakpoints is shorthand for the consecutive full layers between them. A layer can be named with `as`, and the name is used by the structure chart and by a P&L walk. The annual terms `aad` and `agglimit` are described in Using `aad` and `agglimit`.

```
occurrence net of 250 xs 250
occurrence net of 50% po 500 xs 500           # 50% share
occurrence net of 250 po 500 xs 500           # the same: 250 of 500
occurrence ceded to 250 xs 250 and 50% po 500 xs 500 as "Middle"
occurrence ceded to tower [250 500 1000 2000] # 250 xs 250, 500 xs 500, 1000 xs 1000
```

```
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.60.0
```

Net and ceded views of one program

The example book has 20 claims a year from a lognormal severity with mean 100 and coefficient of variation (CV) 3, limited to 2,000. It cedes a program of three layers: 250 xs 250 in full, then 500 xs 500 and 1,000 xs 1,000, each half placed. Written `occurrence net of`, the aggregate is the net loss, and `qd` shows the net severity and the net annual loss.

```
book = '''agg {} 20 claims 2000 xs 0 sev lognorm 100 cv 3
  occurrence {}
    250 xs 250 and 50% po 500 xs 500 and 500 po 1000 xs 1000
  poisson'''
net = build(book.format('Net', 'net of'))
qd(net)
```

```
Aggregate object: Net. Frequency distribution poisson. Severity lognorm,
[0, 2,000], bounded; 2,000 xs 0. Updated with bucket size 1/2 and log2 =
16. Validation: reinsurance; subject not unreasonable; net not
unreasonable.

       Mean     SD      CV    Skew P01 Median    P99
X
Freq     20 4.4721 0.22361 0.22361
Sev  73.469 111.58  1.5188  3.9118   1   31.5  539.5
Agg  1469.4 597.48 0.40662 0.80416 428 1387.5 3199.5
```

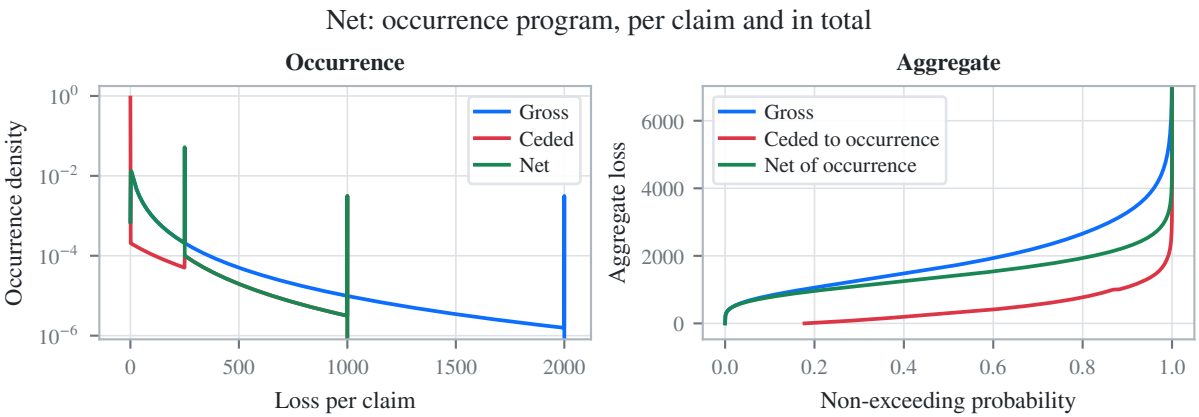
Written `occurrence ceded to`, the same program gives the ceded loss. Both aggregates compute all three views, and `reins_explanation` gives the split of the gross expected loss: 1,901.37 gross, 432.00 ceded, and 1,469.38 net. The ceded mean is the mean of the `ceded to` aggregate.

```
ceded = build(book.format('Ceded', 'ceded to'))
wprint(net.reins_explanation)
print(f'{ceded.est_m:.2f}')
```

```
Net of 100% share of 250 xs 250 and 50% share of 500 xs 500 and 50% share of
1,000 xs 1,000 per occurrence. Per occurrence, gross expected loss 1,901.37
splits into 431.997 ceded (22.7%) and 1,469.38 net.
432.00
```

The method `reins_occ_plot` draws the three views per claim on the left, as densities on a log scale, and for the year on the right, as quantile functions (a Lee diagram). The net claim has an atom at 250, the retention of the full first layer.

```
fig = net.reins_occ_plot()
```



Partial placements and towers

The property `reins_description` shows each layer as a share of a limit in excess of an attachment. A percentage and an amount give the same share, and a `tower` expands to full layers between con-

secutive breakpoints, starting from the first. The descriptions are available before the aggregate is updated.

```
stub = 'agg T 20 claims 2000 xs 0 sev lognorm 100 cv 3 occurrence ceded to {} poisson'
for layer in ['50% po 500 xs 500', '250 po 500 xs 500', 'tower [250 500 1000 2000]']:
    wprint(f'{layer:26s}', build(stub.format(layer), update=False).reins_description)
```

```
50% po 500 xs 500          Ceded to 50% share of 500 xs 500 per occurrence
250 po 500 xs 500          Ceded to 50% share of 500 xs 500 per occurrence
tower [250 500 1000 2000]  Ceded to 100% share of 250 xs 250 and 100% share of
                           500 xs 500 and 100% share of 1,000 xs 1,000 per
                           occurrence
```

A gap between layers can be left as a hole, as in `250 xs 250 and 1000 xs 1000`, or written as a layer with zero share, `0% po 500 xs 500`. A bare amount below 5% of the limit, such as `0.5 po 500 xs 500`, gives a placement of 0.1% and logs a warning suggesting the percentage form.

Reinsurance statistics by layer

The table `reins_stats_df` has one column per layer, `layer.1` to `layer.3`, beside `Gross` and the program totals `Ceded` and `Net`. The `meta` rows give the terms and the probabilities: `pr_attach` is the probability that a claim exceeds the attachment, `pr_detach` the probability that it exhausts the layer, and `lol` (loss on line) the expected annual layer loss over the placed limit. The `freq`, `sev`, and `agg` rows give the moments. The table below shows the main rows, transposed so that each column of the original is a row.

```
rows = [('meta', 'share'), ('meta', 'limit'), ('meta', 'attach'),
        ('meta', 'pr_attach'), ('meta', 'pr_detach'), ('meta', 'lol'),
        ('freq', 'mean'), ('sev', 'mean'), ('agg', 'mean'), ('agg', 'cv')]
s = net.reins_stats_df.droplevel('view', axis=1).loc[rows].T
s.columns = [m if c == 'meta' else f'{c} {m}' for c, m in s.columns]
GT(s, float_format=',.4g')
```

layer	share	limit	attach	pr_attach	pr_detach	lol	freq mean	sev mean	agg mean	agg cv
Gross	1	2,000	0	0.9993	0.003139	0.9507	20	95.07	1,901	0.5296
layer.1	1	250	250	0.08641	0.03445	1.088	1.728	157.5	272.1	0.8825
layer.2	0.5	500	500	0.0344	0.01142	0.3975	0.6881	144.4	99.39	1.435
layer.3	0.5	1,000	1,000	0.01141	0.003139	0.121	0.2283	265.1	60.51	2.552
Ceded	—	1,000	250	0.08641	—	0.432	20	21.6	432	1.066
Net	—	—	—	0.9993	—	—	20	73.47	1,469	0.4066

A layer column is conditional on a claim reaching the layer. The first layer is reached by $20 \times 0.08641 = 1.728$ claims a year, whose mean payment is 157.5, for an expected annual loss of 272.1. The second layer's loss on line is $99.39/(0.5 \times 500) = 0.398$, on the placed limit. The `Ceded` and `Net` columns are unconditional, with the gross claim count, and their severity means, 21.60 and 73.47, add to the gross 95.07. The limit of `Ceded`, 1,000, is the placed capacity $250 + 0.5 \times 500 + 0.5 \times 1000$.

```
st = net.reins_stats_df['occ']
print(f"{20 * st.loc[('meta', 'pr_attach'), 'layer.1']:.4f}",
      f"{st.loc[('freq', 'mean'), 'layer.1'] * st.loc[('sev', 'mean'), 'layer.1']:.2f}")
print(f"{st.loc[('sev', 'mean'), 'Ceded'] + st.loc[('sev', 'mean'), 'Net']:.4f}",
      f"{st.loc[('sev', 'mean'), 'Gross']:.4f}")
```

```
1.7281 272.10
95.0687 95.0687
```

The table `reins_summary_df` compares each view with the gross by component, and `reins_density_df` contains the gross, ceded, and net severities and aggregates on the grid, one column each. The joint law of ceded and net is a bivariate distribution, the subject of Using `grossnet`, `grossceded` and `netceded`.

Aggregate covers and pricing clauses

The same layer clauses after the frequency, as `aggregate net of` or `aggregate ceded to`, apply to the annual loss rather than to each claim. An occurrence program inures to an aggregate cover, so the aggregate layers apply to the output of the occurrence stage.

A layer also accepts a premium, `deposit`, `rol` (rate on line), or `rate` (a fraction of subject premium), and a ceding commission `cede`. They describe the economics of the cession, which only a profit and loss statement can book. A plain `agg` builds the loss structure, drops the pricing clauses with an `IgnoredDecLClauseWarning`, and keeps them in its stored program, so that a `pnl` or `xpnl` referring to it books them: see Using pnl and xpnl.

Refused programs

Layers must be entered bottom up and must not overlap, and a share cannot exceed 100%, whether it comes from a percentage or from an amount above the limit. Occurrence reinsurance acts on each claim, so it cannot be combined with `approximate`, which replaces the aggregate by a fitted distribution.

```
pre = 'agg R 20 claims 2000 xs 0 sev lognorm 100 cv 3 occurrence net of '
refuse(pre + '500 xs 500 and 250 xs 250 poisson',
       pre + '500 xs 250 and 500 xs 500 poisson',
       pre + '150% po 500 xs 500 poisson',
       pre + '600 po 500 xs 500 poisson',
       pre + '250 xs 250 poisson approximate gamma')
```

```
Reinsurance layers must be entered bottom-up (lowest attachment first); got
attachment 250.0 after 500.0. Enter layers in ascending order and use "0 po L xs
A" to represent a gap.
Reinsurance layers overlap: layer attaching at 500.0 starts below the top
(750.0) of the preceding layer. Layers must be non-overlapping; use "0 po L xs
A" to represent a gap.
share 1.5 on '500 xs 500' exceeds 100%; a bare number before po is an amount of
the limit, a percentage is the share
share 1.2 on '500 xs 500' exceeds 100%; a bare number before po is an amount of
the limit, a percentage is the share
DecL: approximate is incompatible with occurrence reinsurance (the
method-of-moments fit bypasses the per-occurrence convolution); use aggregate
reinsurance instead.
```

Using oep

Occurrence exceeding probability and occurrence PML of a Poisson aggregate, compared with aggregate PML (oep)

Stephen J. Mildenhall

2026-11-07

The **occurrence exceeding probability** (OEP) of a loss x is the probability that the largest claim in a year exceeds x, and the loss at a given OEP is an **occurrence PML** (probable maximum loss). The **aggregate exceeding probability** (AEP) is the probability that the year's total loss exceeds x, and the loss at a given AEP is a quantile of the aggregate distribution, the aggregate PML. The total is at least the largest claim, so at every probability the aggregate PML is at least the occurrence PML.

With a Poisson claim count of mean λ and severity survival function $S(x) = \Pr(X > x)$, the claims exceeding x are Poisson with mean $\lambda S(x)$, by thinning. A year has none of them with probability $e^{-\lambda S(x)}$, so

$$\mathrm{OEP}(x) = \Pr(\max_i X_i > x) = 1 - \exp(-\lambda S(x)),$$

and the occurrence PML at annual probability p is $x = S^{-1}(-\log(1-p)/\lambda)$. For a general count with probability generating function $\mathcal{P}_N$, $\Pr(\max_i X_i \le x) = \mathcal{P}_N(1 - S(x))$, and the Poisson formula is the case $\mathcal{P}_N(z) = e^{-\lambda(1-z)}$.

The function `oep(agg, p)`, in `aggregate.utilities` and importable from `aggregate`, computes the occurrence PML of a Poisson aggregate at one or more probabilities p. It returns a table indexed by p with the loss, its severity exceedance `S_sev` and percentile `F_sev`, the achieved OEP, and two return periods: the occurrence return period $1/(\lambda S(x))$, the mean time between claims exceeding x, and the annual return period $1/\mathrm{OEP}(x)$, the mean time between years containing such a claim. The loss is computed from `Aggregate.sev`, the exact continuous severity, so it is not rounded to the grid, and the aggregate need not be updated. The argument `freq` replaces λ. Any count other than a plain Poisson is refused, since the formula depends on thinning.

```
import numpy as np
import pandas as pd
from aggregate import build, oep, __version__
from style import GT, refuse, figures
figures()
print(__version__)
```

```
2.1.0a7
```

Occurrence and aggregate PMLs of a catastrophe book

The example book has two events a year on average, each a single claim, with a lognormal severity of median 1,000 and shape $\sigma = 1$, so mean 1,648.7. The table compares the occurrence PML from `oep` with the aggregate PML, the quantile `q(1 - p)` of the aggregate, at the same annual probabilities.

```
a = build('agg Cat 2 claims sev 1000 * lognorm 1 poisson')
ps = [0.5, 0.2, 0.1, 0.04, 0.02, 0.01, 0.004, 0.001]
o = oep(a, ps)
o['AEP loss'] = [a.q(1 - p) for p in ps]
o['ratio'] = o['AEP loss'] / o['loss']
o.index = pd.Index([f'{p:g}' for p in ps], name='p')
GT(o[['loss', 'occurrence_return_period', 'annual_return_period',
      'AEP loss', 'ratio']], float_format=',.2f')
```

p	loss	occurrence_return_period	annual_return_period	AEP loss	ratio
0.5	1,483.77	1.44	2.00	2,180	1.47
0.2	3,381.14	4.48	5.00	5,330	1.58
0.1	5,050.08	9.49	10.00	7,800	1.54
0.04	7,731.74	24.50	25.00	11,320	1.46
0.02	10,201.78	49.50	50.00	14,270	1.40
0.01	13,119.41	99.50	100.00	17,520	1.34
0.004	17,770.32	249.50	250.00	22,420	1.26
0.001	26,853.23	999.50	1,000.00	31,560	1.18

At 1 in 100 the occurrence PML is 13,119 and the aggregate PML 17,520. Among the probabilities shown, the ratio of the two is largest, 1.58, at 1 in 5, and it falls to 1.18 at 1 in 1,000: the lognormal is subexponential, so a large annual total is usually the result of one large event. The aggregate PMLs are multiples of the bucket size, 10, and the occurrence PMLs are exact. For a continuous severity the achieved `oep` column, omitted here, equals p.

The two return periods differ most at common losses. At $p = 1/2$ a claim above 1,484 occurs every 1.44 years on average, but a year containing one occurs every 2 years, because some years have two or more. For rare losses, with $u = \lambda S(x)$ small, $1/(1-e^{-u}) = 1/u + 1/2 + O(u)$, so the annual return period exceeds the occurrence return period by half a year, 1,000 against 999.5 at 1 in 1,000.

The occurrence exceeding probability as a layer probability

OEP(x) is the probability that an unlimited occurrence layer `inf xs x` pays anything in the year (see Using occurrence reinsurance). The probability is one minus the mass at zero of the ceded aggregate, and `aggregate` computes it for any claim count. The table compares it with the formula for the Poisson count and for a negative binomial count, `mixed gamma 0.5`, a Poisson with a gamma mixing distribution of CV $c = 0.5$, whose generating function is $\mathcal{P}_N(z) = (1 + c^2\lambda(1-z))^{-1/c^2}$.

```
layer = 'agg L 2 claims sev 1000 * lognorm 1 occurrence ceded to inf xs {} {}'
rows, c2 = {}, 0.5 ** 2
for x in [5000, 10000, 20000]:
    s = a.sev.sf(x)
    pr = {m: 1 - build(layer.format(x, m), bs=1, log2=18).density_df.p_total.iloc[0]
          for m in ['poisson', 'mixed gamma 0.5']}
    rows[x] = [pr['poisson'], -np.expm1(-a.n * s),
               pr['mixed gamma 0.5'], 1 - (1 + c2 * a.n * s) ** (-1 / c2)]
df = pd.DataFrame.from_dict(rows, orient='index', columns=[
    'Poisson layer', 'Poisson formula', 'mixed layer', 'mixed formula'])
df.index.name = 'x'
GT(df, float_format='.6f')
```

x	Poisson layer	Poisson formula	mixed layer	mixed formula
5,000	0.101922	0.101942	0.100647	0.100666
10,000	0.021074	0.021077	0.021019	0.021022
20,000	0.002734	0.002734	0.002733	0.002733

The layer and the formula agree to within 0.02% (relative), the error of a grid with buckets of width 1. Mixing lowers the OEP at every x, by Jensen's inequality, since $1 - \mathsf{P}e^{-\Lambda S(x)} \le 1 - e^{-\lambda S(x)}$ for a random Poisson mean Λ with $\mathsf{P}\Lambda = \lambda$. The function `oep` raises an error on the mixed book, and the layer probability is the way to compute its OEP.

Limited severities and the freq argument

On a book with a policy limit the severity has an atom at the limit, and every probability below the annual probability of a claim at the limit gives the limit as the occurrence PML. With a limit of 20,000 that probability is $1 - \exp(-2S(20000)) = 0.002734$. The 1 in 1,000 PML is therefore 20,000, and the achieved `oep`, the probability of a claim strictly above it, is zero, so its return period is infinite (shown as a dash). The 1 in 100 PML is 13,119, the same as without the limit. Passing `freq=4` doubles the event rate, which raises the 1 in 100 PML to 16,534.

```
b = build('agg Lim 2 claims 20000 xs 0 sev 1000 * lognorm 1 poisson')
df = pd.concat({'limit 20,000': oep(b, [0.01, 0.001]),
                'Cat, freq=4': oep(a, 0.01, freq=4)}, names=['book'])
GT(df[['loss', 'oep', 'annual_return_period']], float_format=',.4f')
```

book	*p*	loss	oep	annual_return_period
limit 20,000	0.0100	13,119.4077	0.0100	100.0000
	0.0010	20,000.0000	0.0000	—
Cat, freq=4	0.0100	16,533.9524	0.0100	100.0000

An occurrence program

On an aggregate with an `occurrence net of` or `occurrence ceded to` clause, `oep` returns net or ceded points.[1] The net and ceded losses of a claim are nondecreasing functions of its gross loss, so the largest net or ceded claim in a year is that function of the largest gross claim, and the point is the function applied to the gross point. The example places 10,000 xs 10,000 on the catastrophe book. At 1 in 100 the gross point is 13,119, the net point is the retention of 10,000, and the ceded point is 3,119. Where a function is constant, its loss has an atom, and the achieved OEP falls below p. The net loss is 10,000 for every claim between 10,000 and 20,000, so the 1-in-100 net point is exceeded only by a claim above 20,000, once in 366 years. The ceded loss is 0 below 10,000, so a 1-in-10 ceded point of 0 is exceeded once in 47 years, and the ceded loss is 10,000 above 20,000, so the 1-in-1,000 ceded point is never exceeded.

[1] From aggregate 2.1.0a7

```
df = pd.concat({kind: oep(build(f'agg CatRe 2 claims sev 1000 * lognorm 1 '
                                f'occurrence {kind} 10000 xs 10000 poisson',
                                update=False), [0.1, 0.01, 0.001])
                for kind in ['net of', 'ceded to']}, names=['program'])
GT(df[['loss', 'oep', 'annual_return_period']], float_format=',.4f')
```

program	*p*	loss	oep	annual_return_period
net of	0.1000	5,050.0761	0.1000	10.0000
	0.0100	10,000.0000	0.0027	365.7481
	0.0010	16,853.2270	0.0010	1,000.0000
ceded to	0.1000	0.0000	0.0211	47.4453
	0.0100	3,119.4077	0.0100	100.0000
	0.0010	10,000.0000	0.0000	—

An event loss table

An aggregate built from an event loss table with secondary uncertainty, an `elt` clause with six-column rows and no `no2u` flag (see Using elt), has a severity that is the rate-weighted mixture of the events' beta and two-point laws. The eltq package computes its severity functions exactly from the rows,[2] so `oep` applies to it as to any Poisson aggregate. The table compares the occurrence PMLs of a six-event table with and without secondary uncertainty. With it, the achieved OEP equals *p*. Without it (`no2u`), each event loss is an atom at its mean, so every occurrence PML is one of the six means and the achieved OEP falls below *p*. At 1 in 250 and rarer the PML is the largest mean, 250, which no claim exceeds, so the achieved OEP is zero.

```
table = '''[[1 0.1 5 50 4 2] [2 0.08 12 120 8 5] [3 0.05 25 250 15 10]
           [4 0.03 60 400 30 20] [5 0.02 120 800 50 40]
           [6 0.01 250 1500 100 80]]'''
ps = [0.1, 0.04, 0.01, 0.004, 0.001]
df = pd.concat({flag or 'secondary uncertainty':
                oep(build(f'agg Cat6 elt {flag} {table}', update=False), ps)
                for flag in ['', 'no2u']}, names=['model'])
GT(df[['loss', 'oep', 'annual_return_period']], float_format=',.4f')
```

model	*p*	loss	oep	annual_return_period
secondary uncertainty	0.1000	16.5314	0.1000	10.0000
	0.0400	59.2989	0.0400	25.0000
	0.0100	190.8643	0.0100	100.0000
	0.0040	307.7952	0.0040	250.0000
	0.0010	506.3936	0.0010	1,000.0000
no2u	0.1000	25.0000	0.0582	17.1717
	0.0400	60.0000	0.0296	33.8358
	0.0100	120.0000	0.0100	100.5008
	0.0040	250.0000	0.0000	—
	0.0010	250.0000	0.0000	—

Each evaluation of the severity is a pass over every row of the table, so on a table of a million rows `oep` at 30 probabilities takes a few seconds. The aggregate need not be updated. When it is, the

[2] From aggregate 2.1.0a6

severity quantile on the grid, `q_sev(1 - s)` with $s = -\log(1-p)/\lambda$, is within half a bucket of the exact loss: with buckets of 1/8 it gives 190.875 at 1 in 100, against 190.864 from `oep`.

```
c = build(f'agg Cat6 elt {table}')
s = -np.log1p(-np.array(ps)) / c.n
print(c.bs, [float(c.q_sev(1 - v)) for v in s])
```

```
0.125 [16.5, 59.25, 190.875, 307.75, 506.375]
```

Refused calls

The probability p must lie strictly between 0 and the probability of at least one claim, $1 - e^{-\lambda}$, which is 0.8647 for $\lambda = 2$, since a year with no claims has no largest claim. A negative binomial or a zero-modified Poisson count is refused.

```
nb = build('agg NB 2 claims sev 1000 * lognorm 1 mixed gamma 0.5')
zm = build('agg ZM 2 claims sev 1000 * lognorm 1 poisson zm 0.2')
refuse(lambda: oep(a, 0.9), lambda: oep(a, 0),
       lambda: oep(nb, 0.01), lambda: oep(zm, 0.01))
```

```
p must be below the ceiling 1 - exp(-lam) = 0.8646647 for lam = 2; got [0.9]. A
year with no occurrence has no largest loss, so nothing is exceeded.
p must be in (0, 1) exclusive; got [0.0].
oep requires Poisson frequency; NB has freq_name='gamma'.
oep requires plain Poisson frequency; ZM is zero modified, which breaks the 1 -
exp(-lam S) derivation.
```

Using `pnl` and `xpnl`

Profit and loss statements with the pnl and xpnl kinds and the PnL class

Stephen J. Mildenhall

2026-10-30

An aggregate gives the distribution of loss, but an underwriting result also includes premium and expense, and, with reinsurance, ceded premium, recoveries, and ceding commission. The DecL kinds `pnl` and `xpnl` wrap a complete aggregate or portfolio, which models the loss, and return a `PnL`: the distribution of the underwriting result together with its accounting ledger.

With gross premium P, expense E, gross loss X, ceded premium C, recovery R, and ceding commission K, the net result is

$$M = (P - C) - (X - R) - (E - K).$$

The net result needs only the law of the net loss $X - R$, which the wrapped aggregate computes. A **walk** shows the gross result and the result of each cover separately and adds them outcome by outcome, so it needs the joint law of (X, R); for an occurrence cover, R depends on the individual claims and not only on X. A `PnL` uses the payoff convention: profit is positive, the adverse tail is the left one, and `P01`, the 1% quantile, is the 1 in 100 bad year.

The kinds `pnl` and `xpnl`[1] share one syntax and give two views of the business. A `pnl` is the net position, one group with net premium against net loss and expense. An `xpnl` is the walk, with one group per step: gross first, then the occurrence program, then the aggregate program, and finally `All`, the net.

```
pnl  NAME <premium> less <engine> [less <expenses>]
xpnl NAME <premium> less <engine> [less <expenses>] [peel top-down | peel bottom-up]

<premium>   10000 premium | inherit premium | derive premium      [as "label"]
<engine>    agg NAME <body> | agg.NAME | port NAME <units> | port.NAME
<expenses>  2000 fixed expenses | 0.25 premium expenses | 0.1 loss expenses
```

In the grammar, `<engine>` is the aggregate or portfolio that models the loss, written in full or as a reference to a stored entry. `inherit premium` uses the premium of that aggregate or portfolio, and `derive premium` grosses its premium P_0 up for the expenses, to $(P_0 + F)/(1 - r)$ for fixed expense F and premium expense ratio r. The separator is the keyword `less` rather than `-`, so it cannot be confused with severity arithmetic. In an `xpnl`, `peel` makes each layer of a program its own step, instead of one step per program.

```
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.0

A ceding commission treaty as a pnl and an xpnl

The example is the entry `CedingCommission` in the recipe library included with aggregate (see Using recipes), shortened to `CC`. It is a book with premium 10,000 at a 70% loss ratio and policy limits of 2,000, one occurrence layer 1,750 xs 250 rated at 32.5% of gross premium with a 10% ceding commission, and 2,000 of fixed expense. The program is written once, with a placeholder for the kind, and built first as a `pnl`.

```
program = '''{kind} CC
  inherit premium
  less
    agg CC.Book
      10000 premium at 0.7 lr
      2000 xs 0
      sev lognorm 150 cv 2.5
      occurrence net of
        1750 xs 250 rate 0.325 cede 0.1 as "Occ Re"
      mixed gamma 0.25
  less
    2000 fixed expenses as Expenses'''
p = build(program.format(kind='pnl'))
qd(p)
```

```
PnL('CC': role=sell, 1 consideration, 3 obligation -> 'margin')

                 EX     SD       CV     Skew   P01  Median   P99
Side
Consideration  6750      0        0           6750    6750  6750
Obligation    -6156 1421.4 -0.23089 -0.52485 -9995   -6032 -3410
Margin          594 1421.4   2.3929 -0.52485 -3245     718  3340
```

The `pnl` shows the net position. Net premium 6,750 is the gross 10,000 less 3,250 ceded, and the obligation, 6,156, is net loss plus expense less commission. Built as an `xpnl`, the same program gives the walk: the gross step, the cover step `Occ Re`, then `All`. The cover step ends with the running `Net`, and `All` adds the `Impact`, the margin of the whole reinsurance program.

```
x = build(program.format(kind='xpnl'))
qd(x)
```

```
PnL('CC': 2 groups, 6 legs -> 'margin')

                        EX     SD       CV     Skew    P01  Median   P99
Step   Side
Gross  Consideration 10000      0        0           10000   10000 10000
       Obligation    -9000 2706.5 -0.30072 -0.67617 -16560   -8680 -4120
       Margin         1000 2706.5   2.7065 -0.67617  -6560    1320  5880
Occ Re Consideration -3250      0        0           -3250   -3250 -3250
       Obligation     2844 1619.5  0.56944  0.94221    405    2605  7645
       Margin         -406 1619.5  -3.9889  0.94221  -2845    -645  4395
       Net             594 1426.7   2.4019  -0.5251  -3245     715  3355
All    Consideration  6750      0        0            6750    6750  6750
       Obligation    -6156 1426.7 -0.23176  -0.5251  -9995   -6035 -3395
       Margin          594 1426.7   2.4019  -0.5251  -3245     715  3355
       Impact         -406 1619.5  -3.9889  0.94221  -2845    -645  4395
```

The margin of the cover is −406: the cedant pays 406 a year on average for the protection, and the standard deviation of the result falls from 2,706 to 1,427. The two kinds agree on the mean, 594. The walk is computed from the joint law of gross and ceded loss on a coarser grid, so its standard deviation, 1,427, differs from the 1,421 of the `pnl` in the fourth significant figure. The wrapped

aggregate is the `engine` attribute of a `PnL`, and its mean, 4,481, is the mean net loss. On the `pnl`, `q(0.01)` is the 1 in 100 result, a loss of 3,245, and `cdf(0)`, the probability that the result is at most zero, is 0.312.

```
print(p.est_m, x.est_m)
print(p.est_sd, x.est_sd)
print(p.q(0.01), p.cdf(0), x.engine.est_m)
```

```
593.9985626922165 593.9985628368244
1421.35256663098 1426.7030414293781
-3245.0 0.3122290184639707 4481.00143731014
```

With `derive premium` in place of `inherit premium`, the 2,000 of fixed expense is added to the 10,000 premium of the aggregate, for a gross premium of 12,000. The ceded premium, 32.5% of gross, rises with it to 3,900.

```
d = build(program.format(kind='xpnl').replace('inherit', 'derive'))
print(d.economic_ratios_df.P)
```

```
Step
Gross     12000.0
Occ Re    -3900.0
All        8100.0
Name: P, dtype: float64
```

The accounting ledger

The accounting ledger of a `PnL` is the table `economic_df`, which lists every leg by step and side. Column `κ01` is the conditional mean of each row given that the net result is at its 1% quantile, and likewise `κ50` and `κ99`. Each κ column is one scenario and foots down the table: in the 1 in 100 year the gross margin of −5,380 and the margin of the cover, 2,135, give the net −3,245.

```
GT(x.economic_df[['EX', 'SD', 'κ01', 'κ50', 'κ99']], float_format=',.0f')
```

Step	*Side*	*Label*	EX	SD	κ01	κ50	κ99
Gross	Consideration	Premium	10,000	0	10,000	10,000	10,000
	Obligation	Loss	-7,000	2,706	-13,380	-6,793	-2,469
		Expenses	-2,000	0	-2,000	-2,000	-2,000
		Total	-9,000	2,706	-15,380	-8,793	-4,469
	Margin	Gross	1,000	2,706	-5,380	1,207	5,531
Occ Re	Consideration	Occ Re premium	-3,250	0	-3,250	-3,250	-3,250
	Obligation	Occ Re recovery	2,519	1,619	5,060	2,433	749
		Occ Re commission	325	0	325	325	325
		Total	2,844	1,619	5,385	2,758	1,074
	Margin	Total	-406	1,619	2,135	-492	-2,176
		Net	594	1,427	-3,245	715	3,355
All	Consideration	Net	6,750	0	6,750	6,750	6,750
	Obligation	Net	-6,156	1,427	-9,995	-6,035	-3,395
	Margin	Net	594	1,427	-3,245	715	3,355
		Impact	-406	1,619	2,135	-492	-2,176

Amounts and ratios by step

The table `economic_ratios_df` has one row per step of the walk. It contains premium `P`, loss `L`, expense `E`, margin `M`, and the standard deviation `SD` of the result, signed so that the amounts add across steps. It also contains the loss, expense, and combined ratios `LR`, `ER`, and `CR`. On the cover row `E` is the ceding commission, -325, and on `All` the expense is net of it, $2{,}000 - 325 = 1{,}675$.

```
GT(x.economic_ratios_df[['P', 'L', 'E', 'M', 'SD', 'LR', 'ER', 'CR']],
   float_format=',.0f', ratio_cols=['LR', 'ER', 'CR'])
```

Step	P	L	E	M	SD	LR	ER	CR
Gross	10,000	7,000	2,000	1,000	2,706	70.0%	20.0%	90.0%
Occ Re	-3,250	-2,519	-325	-406	1,619	77.5%	10.0%	87.5%
All	6,750	4,481	1,675	594	1,427	66.4%	24.8%	91.2%

Margin and capital by step

The table `waterfall_df` sets the margin of each step against the capital behind it, at the return period stored in its `attrs`, here 100. The column `Capital standalone` is the negated 1 in 100 result of the step alone; for a cover it comes from the right tail of the cover's own result and is negative, capital released. `Capital net` conditions every step on the whole book at its 1 in 100 result, so it foots to the net capital, $5{,}380 - 2{,}135 = 3{,}245$. `MSD` is margin over its standard deviation, and `CoC net`, the cost of capital, is margin over net capital. On a `pnl` the table is empty, since a single group has no steps.

```
w = x.waterfall_df
print(w.attrs)
GT(w[['Margin', 'MSD', 'Capital standalone', 'Capital net', 'CoC net']],
   float_format=',.0f', formatters={'MSD': '{x:.3f}'}, ratio_cols=['CoC net'])
```

```
{'return_period': 100}
```

Step	Margin	MSD	Capital standalone	Capital net	CoC net
Gross	1,000	0.369	6,560	5,380	18.6%
Occ Re	-406	-0.251	-4,395	-2,135	19.0%
All	594	0.416	3,245	3,245	18.3%

The pentagon of each step at a return period

The **pentagon** of a step is the set of five amounts loss L, margin M, premium $P = L + M$, capital Q, and assets $a = P + Q$. The method `pentagon_df(t)` computes it for each step at return period `t`, with the technical premium `P_tech` $= L + M$ and the capital `Q` equal to the negated 1 in `t` result. It adds the cost of capital `CoC` $= M/Q$, the loss ratio `LR` on written premium, and the technical loss ratio `TLR` on technical premium. The `Ceded` row is the whole cession, with capital the difference between `All` and `Gross`, $3{,}925 - 7{,}960 = -4{,}035$ at a 250 year return period.

```
GT(x.pentagon_df(250)[['L', 'M', 'P_tech', 'Q', 'a', 'CoC', 'LR', 'TLR']],
   float_format=',.0f', ratio_cols=['CoC', 'LR', 'TLR'])
```

Step	L	M	P_tech	Q	a	CoC	LR	TLR
Gross	7,000	1,000	8,000	7,960	15,960	12.6%	70.0%	87.5%
Occ Re	-2,519	-406	-2,925	-5,315	-8,240	7.6%	77.5%	86.1%
All	4,481	594	5,075	3,925	9,000	15.1%	66.4%	88.3%
Ceded	-2,519	-406	-2,925	-4,035	-6,960	10.1%	77.5%	86.1%

Refused programs

Three P&L programs are refused: `inherit premium` on an aggregate with no premium, `peel` on a `pnl`, which has no steps, and `derive premium` with a loss expense, which the gross-up formula does not include. Each error message includes the fix.

```
bare = 'agg Bare 10 claims sev lognorm 10 cv 1 poisson'
priced = 'agg Priced 100 premium at 0.7 lr sev lognorm 10 cv 1 poisson'
refuse(f'pnl R1 inherit premium less {bare}',
       f'pnl R2 100 premium less {priced} peel top-down',
       f'pnl R3 derive premium less {priced} less 0.1 loss expenses')
```

```
R1: 'inherit premium' but the wrapped engine has no premium to inherit -- it
needs a 'premium at lr' exposure (or a stored agg / port carrying premium). Give
an explicit '<amount> premium' instead.
R2: 'peel' is an 'xpnl' clause. A 'pnl' is the consolidated single-group net
view, so it has no steps to peel; write 'xpnl R2 ... peel top-down' for the
layer-by-layer walk.
R3: 'derive premium' cannot gross up a loss basis expense; losses are not
reliably known by inspection. Use fixed or premium expenses, or 'inherit
premium' with the loss expense left in place.
```

Using `pollaczeck_khinchine` and `wiener_hopf`

Eventual ruin probability with pollaczeck_khinchine and wiener_hopf

Stephen J. Mildenhall

2026-11-01

An insurer starts with surplus u, collects premium at rate c, and pays claims X_i. The claims are independent and identically distributed, and so are the waits W_i between them. The surplus is $U(t) = u + ct - \sum_{i \le N(t)} X_i$ and the probability of eventual ruin is $\psi(u) = \Pr(U(t) < 0 \text{ for some } t)$. The premium rate is $c = (1+\rho)\mathsf{P}X/\mathsf{P}W$, where the margin ρ, the ratio of premium margin to expected loss, is also called the safety loading. Ruin can only happen at a claim, so $\psi(u) = \Pr(M > u)$, where M is the all-time maximum of the random walk with step $Y = X - cW$, finite when $\rho > 0$.

For a Poisson count (the Cramér-Lundberg model), M is a geometric sum: the walk rises above its running maximum again with probability $1/(1+\rho)$, and each rise, or ladder height, has the equilibrium law $F_I(x) = \int_0^x \Pr(X > y)\, dy/\mathsf{P}X$. The result is the Pollaczeck-Khinchine formula

$$1 - \psi(u) = \frac{\rho}{1+\rho} \sum_{n \ge 0} (1+\rho)^{-n} F_I^{*n}(u),$$

which `Aggregate.pollaczeck_khinchine(rho)` evaluates by FFT on the severity grid. For a renewal count (the Sparre-Andersen model), M is still a geometric sum of ladder heights, but neither the probability $1/(1+\rho)$ nor the equilibrium law holds. The method `Aggregate.wiener_hopf(rho)` computes the ladder structure numerically, by a cepstral Wiener-Hopf factorization of the law of Y. Renewal counts are described in Using renewal counts.

Exponential claims with mean m give closed forms. With Poisson arrivals $\psi(u) = e^{-\rho u/((1+\rho)m)}/(1+\rho)$. With any wait law $\psi(u) = (1-Rm)e^{-Ru}$, where the adjustment coefficient, or Lundberg exponent, $R > 0$ solves $\mathsf{P}e^{RX}\,\mathsf{P}e^{-cRW} = 1$.

The method `pollaczeck_khinchine` requires a Poisson count, and the method `wiener_hopf`[1] requires a renewal count. Both return the named tuple `RuinFunction(ruin, find_u, mean, density)`. The examples use three aggregates with exponential claims of mean 1.

```
agg Po 10 claims sev expon poisson        # pollaczeck_khinchine
agg Re 10 years sev expon wait expon      # wiener_hopf, the same model
agg Ga 10 years sev expon wait gamma 2    # wiener_hopf, regular arrivals
```

```
import numpy as np
import pandas as pd
from scipy.optimize import brentq
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.60.0
```

[1] From aggregate 1.0

Ruin probability with Poisson arrivals and exponential claims

For Poisson arrivals and exponential claims with mean 1, the `ruin` field of the result of `pollaczeck_khinchine` is $\psi(u)$ as a Series on the severity grid. The same model written as a renewal count with exponential waits is solved by `wiener_hopf`. At $\rho = 0.2$ both solvers agree with the closed form to within 4×10^{-4}, and `wiener_hopf` is the closer of the two on the same grid.

```
rho = 0.2
po = build('agg Po 10 claims sev expon poisson')
re = build('agg Re 10 years sev expon wait expon')
pk = po.pollaczeck_khinchine(rho)
wh = re.wiener_hopf(rho)
u = np.array([0, 1, 2, 5, 10, 20, 40])
df = pd.DataFrame({'PK': pk.ruin.loc[u].values, 'WH': wh.ruin.loc[u].values,
                   'exact': np.exp(-rho * u / (1 + rho)) / (1 + rho)},
                  index=pd.Index(u, name='u'))
GT(df, float_format='.5f')
```

u	PK	WH	exact
0	0.83306	0.83320	0.83333
1	0.70508	0.70529	0.70540
2	0.59675	0.59701	0.59711
5	0.36180	0.36211	0.36217
10	0.15713	0.15737	0.15740
20	0.02964	0.02972	0.02973
40	0.00105	0.00106	0.00106

The function `find_u(p)` returns the surplus with ruin probability p, snapped to the grid under the default `kind='index'` and interpolated under any other `kind`. The exact surplus is $-(1+\rho)m\log((1+\rho)p)/\rho$, which for $p = 1\%$ is 26.5371, the value from `wiener_hopf`. The field `mean` is the discretized severity mean, here 0.9999998.

```
for p in [0.05, 0.01]:
    print(f'{p:.2f} {pk.find_u(p):8.4f} {wh.find_u(p):8.4f} '
          f'{-(1 + rho) * np.log((1 + rho) * p) / rho:8.4f}')
print(pk.mean, wh.mean)
```

```
0.05  16.8652  16.8809  16.8805
0.01  26.5137  26.5371  26.5371
0.9999998410542983 0.9999998410542983
```

Gamma waits and the adjustment coefficient

Gamma waits with shape 2, mean 2, and coefficient of variation (CV) $1/\sqrt{2}$ make arrivals more regular than Poisson arrivals. With exponential claims of mean 1 and $\rho = 0.2$, the premium rate is $c = 1.2 \times 1/2 = 0.6$, and the adjustment coefficient R solves $(1-R)^{-1}(1+cR)^{-2} = 1$. Here $R = 0.2178$, and `wiener_hopf` agrees with the closed form $(1-R)e^{-Ru}$ to within 2×10^{-4}.

```
ga = build('agg Ga 10 years sev expon wait gamma 2')
wg = ga.wiener_hopf(rho)
c = (1 + rho) * 1 / 2
R = brentq(lambda r: (1 + c * r) ** -2 / (1 - r) - 1, 1e-9, 1 - 1e-9)
df = pd.DataFrame({'WH': wg.ruin.loc[u].values, 'exact': (1 - R) * np.exp(-R * u)},
                  index=pd.Index(u, name='u'))
print(f'R = {R:.6f}')
GT(df, float_format='.5f')
```

```
R = 0.217771
```

u	WH	exact
0	0.78206	0.78223
1	0.62902	0.62915
2	0.50593	0.50604
5	0.26324	0.26330
10	0.08861	0.08863
20	0.01004	0.01004
40	0.00013	0.00013

Ruin probability for regular and bursty arrivals

At the same margin, $\rho = 0.2$, the regularity of arrivals changes the ruin probability a great deal. Every wait law in the table has mean one, and the claims are exponential with mean 1. The surplus for a 1% ruin probability runs from about 14 for gamma waits with CV 1/4 to about 39 for a hyperexponential mixture of exponentials with means 1/2 and 2 and CV $\sqrt{2}$.

```
waits = {'gamma cv 0.25': 'gamma 1 cv 0.25', 'gamma cv 0.5': 'gamma 1 cv 0.5',
         'exponential': 'expon', 'hyperexponential': '[0.5 2] * expon wts [2/3 1/3]'}
rows = {}
for k, w in waits.items():
    rf = build(f'agg W 10 years sev expon wait {w}').wiener_hopf(rho)
    rows[k] = {'psi(0)': rf.ruin.iloc[0], 'psi(10)': rf.ruin.loc[10.0],
               'u at 1%': rf.find_u(0.01)}
GT(pd.DataFrame(rows).T.rename_axis('wait law'))
```

wait law	psi(0)	psi(10)	u at 1%
gamma cv 0.25	0.7025	0.0359	14.2998
gamma cv 0.5	0.7428	0.0568	16.7539
exponential	0.8332	0.1574	26.5371
hyperexponential	0.8841	0.2776	38.6973

The method `wiener_hopf` returns ψ on the severity grid. A heavy-tailed severity or a bursty wait law can leave probability for M beyond the end of the grid, and the method then issues a warning giving a larger `log2` to pass, as in `wiener_hopf(rho, log2=...)`. To evaluate ψ at a single capital level, with a check by simulation, use `Aggregate.eventual_ruin`, described in Using eventual_ruin.

Refused calls

Each solver refuses the other's count, and any other count, such as the negative binomial. The method `wiener_hopf` also refuses a defective wait law, under which claims eventually stop arriving, and a margin that fails the net profit condition on the grid. Both solvers refuse a margin $\rho \leq 0$, at which ruin is certain.

```
nb = build('agg Nb 10 claims sev expon negbin 0.5')
de = build('agg De 2 years dsev [1] dwait [1 2] [0.4 0.5] !')
refuse(lambda: re.pollaczeck_khinchine(rho), lambda: po.wiener_hopf(rho),
       lambda: nb.pollaczeck_khinchine(rho), lambda: de.wiener_hopf(rho),
       lambda: re.wiener_hopf(-0.1), lambda: po.pollaczeck_khinchine(0))
```

```
pollaczeck_khinchine assumes a Poisson frequency, got 'renewal'; for a renewal
(wait-clause) frequency use wiener_hopf
wiener_hopf requires a renewal (wait-clause) frequency, got 'poisson'; for a
Poisson frequency use pollaczeck_khinchine
pollaczeck_khinchine assumes a Poisson frequency, got 'negbin'
wiener_hopf requires a proper waiting-time law; this one is defective
(terminating renewal process), wait_defect = 0.1
net profit condition fails on the grid: mean per-claim step 0.1 >= 0; increase
the margin rho
net profit condition requires rho > 0, got 0
```

Using `price`

Distorted ask and bid prices of a loss distribution with Distortion.price and Aggregate.price

Stephen J. Mildenhall

2026-11-06

A distortion g (see Using distortions) prices a loss $X \ge 0$ supported by assets a as

$$g[X \wedge a] = \int_0^a g(S(x))\,dx,$$

where S is the survival function of X. Square brackets mark g acting on a random variable, as a spectral risk measure, and parentheses mark g acting on a probability. Losses above a are not paid, so the price is that of $X \wedge a$, and $a = \infty$ prices the whole loss. This price is the **ask**, the premium for taking the loss on. The **bid** uses the dual $\check{g}(s) = 1 - g(1 - s)$. It equals $-g[-(X \wedge a)]$, the most a party pricing with g would pay to receive $X \wedge a$, and for a concave g

$$\check{g}[X \wedge a] \le \mathsf{P}(X \wedge a) \le g[X \wedge a].$$

For a discrete X with outcomes $x_0 < x_1 < \cdots$ the integral is a sum. The distorted probability of x_k is $q_k = g(\Pr(X \ge x_k)) - g(\Pr(X > x_k))$, and $g[X \wedge a] = \sum_k \min(x_k, a)\, q_k$, an expectation under the probabilities q. The library computes the price this way, exactly, on the outcomes of the distribution.

The price is available in two ways. The method `Distortion.price(ser, a=np.inf, kind='ask')` prices any distribution given as a pandas Series of probabilities indexed by outcome. Its `kind` is `'ask'`, `'bid'`, or `'both'`, and it returns a named tuple `(bid, el, ask)`, or a one-row frame with `as_frame=True`; the asset level must be one of the outcomes. The method `Aggregate.price(p, g, kind='var')` prices an aggregate with assets equal to its VaR at probability `p`, or its TVaR with `kind='tvar'`. It returns the pricing **pentagon**: expected loss L, margin M, premium $P = L + M$, capital Q, and assets $a = P + Q$, with the loss ratio $\mathtt{LR} = L/P$, the premium to capital ratio $\mathtt{PQ} = P/Q$, and the return on capital $\mathtt{ROE} = M/Q$. The examples use one aggregate and one distortion.

```
agg Book 25 claims 2000 xs 0 sev lognorm 100 cv 4 poisson
distortion Gentle ph 0.6
```

```
import numpy as np
import pandas as pd
from aggregate import build, qd, Distortion, __version__
from aggregate.spectral import consistent_distortions
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
2.1.0a5
```

A three-point loss priced by hand

A loss of 0, 100, or 1,000, with probabilities 0.9, 0.09, and 0.01, can be priced by hand. Its survival function is 0.1 on $[0, 100)$ and 0.01 on $[100, 1000)$, so with $g(s) = s^{0.6}$ the ask is $100\,g(0.1) + 900\,g(0.01) = 25.1 + 56.8 = 81.9$, against an expected loss of 19. The bid is $100\,\check{g}(0.1) + 900\,\check{g}(0.01) = 6.1 + 5.4 = 11.5$. With assets of 100 only the first layer is priced: the ask is 25.1, the expected loss 10, and the bid 6.1.

```
gentle = build('distortion Gentle ph 0.6')
x = pd.Series([0.9, 0.09, 0.01], index=[0, 100, 1000])
print(gentle.price(x, kind='both'))
print(gentle.price(x, a=100, kind='both'))
```

```
Price(bid=11.536811368514599, el=19.000000000000007, ask=81.90502531831322)
Price(bid=6.1259606640430615, el=9.999999999999996, ask=25.1188643150958)
```

Prices of an aggregate with Distortion.price

The aggregate `Book`, used here and on Using `price` and `evaluate`, has 25 claims a year, Poisson, from a lognormal severity with mean 100 and CV 4 under a policy limit of 2,000. The limited severity has mean 91.3 and the aggregate mean 2,283.7.

```
book = build('agg Book 25 claims 2000 xs 0 sev lognorm 100 cv 4 poisson')
qd(book)
```

```
Aggregate object: Book. Frequency distribution poisson. Severity
lognorm, [0, 2,000], bounded; 2,000 xs 0. Updated with bucket size 1 and
log2 = 16. Validation: not unreasonable.

       Mean     SD      CV   Skew P01 Median  P99
X
Freq     25      5     0.2    0.2
Sev  91.349 218.98  2.3971 5.5614   0     24 1217
Agg  2283.7 1186.3 0.51947 1.0825 534   2036 5903
```

Any distribution in a Series can be priced, and the column `p_total` of `density_df` is the distribution of the aggregate. Unlimited, the ask is 2,994.9 against a mean of 2,283.7. With assets at the 99.5% VaR, $a = 6{,}438$, the limited expected loss is 2,280.1, the ask 2,945.6, and the bid 1,833.3.

```
ser = book.density_df.p_total
a = book.q(0.995)
GT(pd.DataFrame([gentle.price(ser, kind='both'), gentle.price(ser, a=a, kind='both')],
                index=pd.Index(['unlimited', f'{a:,.0f}'], name='assets')),
   float_format=',.1f')
```

assets	bid	el	ask
unlimited	1,835.5	2,283.7	2,994.9
6,438	1,833.3	2,280.1	2,945.6

Prices of an aggregate with Aggregate.price

The method `Aggregate.price` takes a probability `p` or an asset level `a=`, one of them, and returns the whole pentagon. At `p = 0.995` the assets are 6,438, the 99.5% VaR, and the premium P is the same as the ask from `Distortion.price` at those assets. The margin of 665.5 is a return of 19.1% on capital of 3,492.4, at a loss ratio of 77.4%. With `kind='tvar'` the assets are the 99.5% TVaR, 7,166 on the grid, and the larger capital lowers the return to 16.3%.

```
price = pd.concat([book.price(0.995, gentle, kind=k).rename(index={'Book': k})
                   for k in ['var', 'tvar']])
print(price.loc['var', 'P'] - gentle.price(ser, a=a).ask)
GT(price, float_format=',.1f', ratio_cols=['LR', 'PQ', 'ROE'])
```

```
-5.9117155615240335e-12
```

unit	L	M	P	Q	a	LR	PQ	ROE
var	2,280.1	665.5	2,945.6	3,492.4	6,438	77.4%	84.3%	19.1%
tvar	2,282.4	686.2	2,968.6	4,197.4	7,166	76.9%	70.7%	16.3%

A distortion with a mass at zero prices at any finite level, even on a book with unbounded losses, because $X \wedge a$ is bounded and the mass lands at a. The constant cost of capital gives $P = (L + ra)/(1 + r)$, and the two numbers below are the premium and that formula.

```
coc = Distortion('ccoc', r=0.1)
row = book.price(g=coc, a=a).iloc[0]
print(row.P, (row.L + 0.1 * a) / 1.1)
```

```
2658.0794379107156 2658.0794379111408
```

Five distortions with one Gini index

The five distortions with gini $= 0.25$ from `consistent_distortions` have the same index and price the book differently. With assets 6,438 the ask runs from 2,695.8 for TVaR to 3,319.6 for `ccoc`, in the order of the weight each puts on the extreme tail. For `ccoc` the ask is $(1 - d)L + d\,a$ and the bid $(1 - d)L$, with $d = 0.25$: the expected loss moved a quarter of the way to the assets, or to the smallest loss, zero.

```
rows = {k: [d.gini, d.label, *d.price(ser, a=a, kind='both')]
        for k, d in consistent_distortions(0.25).items()}
GT(pd.DataFrame.from_dict(rows, orient='index',
                          columns=['gini', 'label', 'bid', 'el', 'ask']).rename_axis('kind'),
   float_format=',.1f', formatters={'gini': '{x:.3f}'})
```

kind	gini	label	bid	el	ask
ccoc	0.250	ccoc(0.333)	1,710.1	2,280.1	3,319.6
ph	0.250	PH(0.6)	1,833.3	2,280.1	2,945.6
wang	0.250	Wang(0.451)	1,810.2	2,280.1	2,830.2
dual	0.250	dual(1.67)	1,780.1	2,280.1	2,750.8
tvar	0.250	TVaR(0.25)	1,731.9	2,280.1	2,695.8

Refused calls

The method `Distortion.price` refuses an asset level that is not an outcome, since it does not interpolate; a `kind` other than ask, bid, or both; and probabilities passed as an array without their outcomes. The method `Aggregate.price` refuses a `p` above 1, which is not a probability, and `p = 1` on a book with unbounded losses, which has no maximum. The method `apply_distortion` has no asset level, so it refuses a mass distortion on such a book.

```
refuse(lambda: gentle.price(ser, a=6438.5),
       lambda: gentle.price(ser, kind='mid'),
       lambda: gentle.price(ser.to_numpy()),
       lambda: book.price(1.5, gentle),
       lambda: book.price(1, gentle),
       lambda: book.apply_distortion(coc))
```

```
a=6438.5 must be in the index of ser
kind must be 'bid', 'ask', or 'both'
ser must be a pandas Series, not <class 'numpy.ndarray'>
price: p must be a probability of at most 1, got 1.5; pass an asset level as
a=1.5
price: Book is unbounded, so p=1 does not fix an asset level. It resolves to the
top of the FFT grid, which moves with log2 rather than with the risk, and every
number priced off it moves with it. Pass a= to set the level (a=Book.q(1) is the
top of the grid), use a p below 1, or certify the support with Book.bounded =
True. tail_behavior_df shows the tail.
mass distortion (ccoc(0.1)) on an unbounded aggregate: the mass lands on the
last represented bucket, a different bounded problem. Use price with p below 1
or a= for a finite asset level, or certify `bounded = True` if the support is in
fact bounded.
```

Using `price` and `evaluate`

Premiums from distortions and breakeven distortions from premiums on an Aggregate with price, price_pentagon and evaluate

Stephen J. Mildenhall

2026-11-06

Pricing and evaluation solve the same equation for different unknowns. With assets a behind a loss X, a distortion g and a premium P are consistent when

$$g[X \wedge a] = \int_0^a g(S(x))\,dx = P.$$

Square brackets mark g acting on a random variable, as a spectral risk measure, and parentheses mark g acting on a probability. Pricing fixes g and solves for P, which is what `Aggregate.price(p, g)` does, with a the VaR at probability p (see Using `price`). Evaluation fixes P and solves, for each family of distortions, for the shape that satisfies the equation. The position $P - X \wedge a$ is acceptable to every distortion in the family up to that shape, so the **breakeven** distortion is the most stress the position survives, the acceptability level of Cherny and Madan. Its gini (see Using distortions) is an acceptability index on one scale for every family: the larger it is, the more stress the premium allows for.

Both answers come with the accounting of the **pentagon**: expected loss $L = \mathsf{P}(X \wedge a)$, margin $M = P - L$, premium P, capital $Q = a - P$, and assets a, with the ratios $\mathsf{LR} = L/P$, $\mathsf{PQ} = P/Q$, and $\mathsf{ROE} = M/Q$. The method `Aggregate.price_pentagon` completes the pentagon from the assets, given by `p=` or `a=`, and one target, one of `P`, `M`, `Q`, `LR`, `PQ`, and `ROE`. No distortion is involved: the assets fix L, and the identities do the rest.

The method `Aggregate.evaluate(P, p=None, a=None)` returns an `EvaluationResult`, whose `evaluation_df` has one row per family. With `p=` or `a=` the position is $P - X \wedge a$ and the families are `ccoc`, `ph`, `wang`, `dual`, and `tvar`. Without either, the position is $P - X$ against the whole distribution, and `ccoc`, which needs an asset level, is left out. The argument `P` defaults to the premium declared in the DecL program. The examples use the same book as Using `price`.

```
agg Book 25 claims 2000 xs 0 sev lognorm 100 cv 4 poisson
distortion Gentle ph 0.6
```

```
import warnings
import pandas as pd
from aggregate import build, Distortion, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
book = build('agg Book 25 claims 2000 xs 0 sev lognorm 100 cv 4 poisson')
gentle = build('distortion Gentle ph 0.6')
```

```
2.1.0a5
```

The premium and pentagon from a distortion

At $p = 0.995$ the assets are 6,438, and the method `price` with $g(s) = s^{0.6}$ gives a premium of 2,945.6, a return of 19.1% on capital. The method `price_pentagon` with the same assets and that premium returns the same pentagon from the identities alone. Given a cost of capital of 15% in place of the premium, it returns the premium that earns it, 2,822.4, which is $L + (a - L)\, r/(1 + r)$ with $r = 0.15$.

```
priced = book.price(0.995, gentle)
P, L = priced.loc['Book', ['P', 'L']]
pent = pd.concat([priced.rename(index={'Book': 'price'}),
                  book.price_pentagon(p=0.995, P=P).rename(index={'total': 'price_pentagon(P)'}),
                  book.price_pentagon(p=0.995, ROE=0.15).rename(index={'total':
'price_pentagon(ROE)'})])
GT(pent, float_format=',.1f', ratio_cols=['LR', 'PQ', 'ROE'])
```

unit	L	M	P	Q	a	LR	PQ	ROE
price	2,280.1	665.5	2,945.6	3,492.4	6,438	77.4%	84.3%	19.1%
price_pentagon(P)	2,280.1	665.5	2,945.6	3,492.4	6,438	77.4%	84.3%	19.1%
price_pentagon(ROE)	2,280.1	542.3	2,822.4	3,615.6	6,438	80.8%	78.1%	15.0%

Breakeven distortions from a premium

Evaluating the premium of 2,945.6 at the same assets reverses the pricing calculation: the `ph` row recovers the shape $a = 0.6$, to a solver error of 10^{-10}. The `ccoc` row recovers the return of 19.1%, since `ccoc` at return r prices to exactly the premium that earns r on capital. The other rows are the shapes of the other families that give the same premium. Their indexes run from 0.160 for `ccoc` to 0.386 for `tvar`: the families differ in how much stress the same premium allows for, and the index is lowest for the families that load the tail.

```
ev = book.evaluate(P, p=0.995)
GT(ev.evaluation_df.loc['Book', ['gini', 'param_name', 'param', 'error', 'status']],
   formatters={'gini': '{x:.3f}', 'param': '{x:.4f}', 'error': '{x:.1e}'})
```

distortion	gini	param_name	param	error	status
ccoc	0.160	r	0.1906	0.0e+00	ok
ph	0.250	a	0.6000	1.4e-10	ok
wang	0.296	lam	0.5382	6.4e-08	ok
dual	0.344	b	2.0510	-9.1e-13	ok
tvar	0.386	p	0.3857	7.4e-06	ok

Each breakeven distortion prices the book back to the premium. The check uses `Distortion.price` on the distribution of the book, at the same assets.

```
ser = book.density_df.p_total
for k, row in ev.evaluation_df.loc['Book'].iterrows():
    d = Distortion(k, **{row.param_name: row.param})
    print(f'{d.label:12s} {d.price(ser, a=ev.a).ask:,.2f}')
```

```
ccoc(0.191)  2,945.63
PH(0.6)      2,945.63
Wang(0.538)  2,945.63
dual(2.05)   2,945.63
TVaR(0.386)  2,945.63
```

The same equation is solved in three other places. The `tvar` breakeven level, 0.3857, is the threshold `p_star` of the `Bounds` class, which computes the envelope of every distortion that prices the book to the premium (see Using `Bounds` and Using `PricingBounds`). The method `calibrate_distortions(coc, p=)` solves the same equation with the premium set by a cost of capital. The method `PnL.evaluate` evaluates each step of a profit and loss walk, including one whose consideration is random, such as a swing-rated premium (see Using pnl and xpnl).

Evaluation with and without assets

Without `p=` or `a=` the position is the premium against the unlimited loss, which has the tail beyond 6,438 to pay for, so the same premium allows for less stress. The `ph` breakeven rises from 0.600 to 0.618 and its index falls from 0.250 to 0.236. The panel `a = 6,438` is the one to compare with a calibration at the same assets.

```
both = pd.concat({'a = 6,438': ev.evaluation_df.loc['Book', ['gini', 'param']],
                  'unlimited': book.evaluate(P).evaluation_df.loc['Book', ['gini', 'param']]},
                 axis=1)
GT(both, float_format='.4f')
```

	a = 6,438		unlimited	
distortion	**gini**	**param**	**gini**	**param**
ccoc	0.1601	0.1906	—	—
ph	0.2500	0.6000	0.2358	0.6184
wang	0.2965	0.5382	0.2897	0.5253
dual	0.3445	2.0510	0.3410	2.0350
tvar	0.3857	0.3857	0.3827	0.3827

The Gini index across premiums

A premium set by a loss ratio, $P = L/\text{LR}$, is evaluated below at loss ratios from 90% down to 70%. Every index rises as the loss ratio falls, and the families keep their order at every premium, from `ccoc` lowest to `tvar` highest. At a 75% loss ratio the indexes run from 0.183 to 0.432. The table puts premiums down and families across, with one index in each cell.

```
scan = {f'{lr:.0%}': book.evaluate(L / lr, p=0.995).evaluation_df.loc['Book', 'gini']
        for lr in [0.9, 0.85, 0.8, 0.75, 0.7]}
GT(pd.DataFrame(scan).T.rename_axis('LR'), float_format='.3f')
```

LR	ccoc	ph	wang	dual	tvar
90%	0.061	0.104	0.121	0.138	0.153
85%	0.097	0.160	0.187	0.215	0.242
80%	0.137	0.218	0.258	0.299	0.336
75%	0.183	0.280	0.333	0.388	0.432
70%	0.235	0.345	0.412	0.481	0.529

A premium below the expected loss

A premium below the expected loss, 2,280.1, has no breakeven, since every concave distortion prices the loss at or above its mean, and so above the premium. The method `evaluate` returns NaN with the reason in `status`, and issues one `DegenerateEvaluationWarning` for the whole panel.

```
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter('always')
    low = book.evaluate(2200, p=0.995)
wprint(caught[0].message)
GT(low.evaluation_df.loc['Book', ['gini', 'param', 'status']])
```

```
no breakeven acceptability level for Book, reporting NaN (Book: E[M] = -80.0874
<= 0).
```

distortion	gini	param	status
ccoc	—	—	E[M] = -80.0874 <= 0
ph	—	—	E[M] = -80.0874 <= 0
wang	—	—	E[M] = -80.0874 <= 0
dual	—	—	E[M] = -80.0874 <= 0
tvar	—	—	E[M] = -80.0874 <= 0

Refused calls

Four calls are refused: `evaluate` on a book with no premium declared and none passed; `p=1` on a book with unbounded losses, where it does not determine an asset level; both `p=` and `a=`; and `price_pentagon` with two targets.

```
refuse(lambda: book.evaluate(),
       lambda: book.evaluate(P, p=1),
       lambda: book.evaluate(P, p=0.995, a=6438),
       lambda: book.price_pentagon(p=0.995, P=P, ROE=0.15))
```

```
Book has no premium to evaluate against: pass P=, or declare one in DecL (e.g.
"1000 premium at 0.7 lr").
evaluate: Book is unbounded, so p=1 does not fix an asset level. It resolves to
the top of the FFT grid, which moves with log2 rather than with the risk, and
every number priced off it moves with it. Pass a= to set the level (a=Book.q(1)
is the top of the grid), use a p below 1, or certify the support with
Book.bounded = True. tail_behavior_df shows the tail.
evaluate: pass at most one of p= or a=.
price_pentagon: pass exactly one pricing target (one of P, M, Q, LR, PQ, ROE);
got 2.
```

Using `PricingBounds`

Exact price ranges of other risks over all distortions that price a reference risk to a given premium

Stephen J. Mildenhall

2026-11-06

When a portfolio X with assets a is priced to a premium P, the pricing distortion is not known, only that it is one of the set G_P of distortions g with $g[X \wedge a] = P$. Square brackets mark g acting on a random variable, as a spectral risk measure, and parentheses mark g acting on a probability. The class `PricingBounds`[1] computes the range of the price $g[Y]$ of another risk Y, such as a reinsurance layer, over every g in G_P. The narrower the range, the more closely the price of X determines the price of Y.

A distortion g with Kusuoka measure μ prices any risk Z by $g[Z] = \int \mathrm{TVaR}_p(Z)\,\mu(dp)$. The premium of X and the price of Y are both linear in μ, so the extreme prices are attained by **biTVaRs**, mixtures of two TVaRs. As p runs over $[0,1]$ the point $(\mathrm{TVaR}_p(X), \mathrm{TVaR}_p(Y))$ traces a curve, a biTVaR prices X and Y at a point on a chord of the curve, and the range of the price of Y is the vertical slice at $\mathrm{TVaR}_X = P$ through the convex hull of the curve. Between consecutive CDF breakpoints of X and Y both TVaRs are affine in $1/(1-p)$, so on the discrete grid the curve is exactly piecewise linear, with vertices at the union of the breakpoints, and the bounds are exact. With finite a both risks are capped at a.

The envelope of the distortions in G_P, computed by `Bounds` (Using `Bounds`), also bounds the price of Y, but not sharply, and `PricingBounds` computes the exact range. It is a subclass of the same convex hull engine as `AllocationBounds` (Using `AllocationBounds`), which bounds the share of P allocated to each unit of a portfolio. Use `PricingBounds` for the price of a risk on its own, `AllocationBounds` for a unit's share of the portfolio premium, and `Bounds` for the distortions themselves. The hull is computed once, and `bounds(P)` slices it at any premium in `premium_range`.

```
from aggregate.bounds import PricingBounds
PricingBounds(x_source, y_sources, a=np.inf, s_floor=1e-14)
port.pricing_bounds(y_sources, a=0, p=0)     # the total of port is x_source
```

The reference `x_source` is an `Aggregate`, a `Portfolio` (its total), a `PnL` (its loss engine), a probability `Series` indexed by outcome, or `'uniform'`. The argument `y_sources` is one such risk, a list of them, or a dictionary from names to risks. The method `Portfolio.pricing_bounds` sets a from `a`, or from a probability `p` as $a = q(p)$, and with both zero leaves the risks uncapped.

```
import numpy as np
import pandas as pd
from aggregate import build, __version__
from aggregate.bounds import PricingBounds
from style import wprint, GT, refuse, figures, figure
```

[1] From aggregate 1.0

```
figures()
print(__version__)
```

```
2.1.0a5
```

Aggregate layers on a portfolio

The example is the portfolio `Two` of Using Bounds. Unit `Thin` has 10 claims with a gamma severity of mean 10 and CV 0.5, and unit `Thick` has 2 claims with a lognormal severity of mean 20 and CV 2, limited to 250. Assets are the 99th percentile of the total, $a = 365.125$, and the premium $P = 158.797$ earns a 10% cost of capital. The risks to price are two aggregate layers on the total, 50 xs 150 and 100 xs 250, built as probability series from `density_df`, and the total itself, under the name `Two`.

```
port = build('''port Two
  agg Thin  10 claims sev gamma 10 cv 0.5 poisson
  agg Thick  2 claims 250 xs 0 sev lognorm 20 cv 2 poisson''')
a = port.q(0.99)
port.calibrate_distortions(0.10, a=a)
P = port.calibration_df.P.iloc[0]
x = port.density_df.loss
layers = {f'{l} xs {r}': port.density_df.p_total.groupby(np.clip(x - r, 0, l)).sum()
          for r, l in [(150, 50), (250, 100)]}
pb = PricingBounds(port, {**layers, 'Two': port}, a=a)
wprint(pb)
GT(pb.bounds(P).droplevel('P'), float_format='.3f')
```

```
PricingBounds(X='X', Y=['50 xs 150', '100 xs 250', 'Two'], a=365.125, 8243
vertices, premium range [138.164, 365.125])
```

risk	lower	upper	width
100 xs 250	3.810	11.811	8.000
50 xs 150	14.098	18.497	4.399
Two	158.797	158.797	0.000

The feasible premiums run from $L = \mathsf{P}(X \wedge a) = 138.164$ to a. At P the price of 50 xs 150 lies between 14.098 and 18.497, and the price of 100 xs 250 between 3.810 and 11.811, an upper end about three times the lower. The total has the single price P, because every distortion in G_P prices it to P.

Each of the five calibrated distortions (see Using distortions) is in G_P, so its price of each layer lies in the range. The table lists them in order of their Gini index, `gini`. The `tvar` distortion attains the lower bound on 100 xs 250. On 50 xs 150 the largest calibrated price, 16.245 from `ph`, is well short of the upper bound.

```
prices = pd.DataFrame({n: {k: d.price(y).ask for k, d in port.distortions.items()}
                       for n, y in layers.items()})
prices.insert(0, 'gini', port.distortion_df.gini)
GT(prices.rename_axis('distortion'), float_format='.3f', formatters={'gini': '{x:.4f}'})
```

distortion	gini	50 xs 150	100 xs 250
ccoc	0.0909	14.405	11.702
ph	0.1556	16.245	7.359
wang	0.1868	16.218	5.813
dual	0.2196	15.791	4.447
tvar	0.2461	14.386	3.810

Distortions that attain the pricing bounds

The method `bitvars(P)` lists the biTVaR at each end of each range, as `p0`, `p1`, and the weight `w1` on `p1`. The upper bound on 100 xs 250 mixes the mean, $p_0 = 0$, with TVaR at 0.985, with weight 0.093. It is close to the `ccoc` distortion, which mixes the mean with the maximum. The lower bound is a bracket around $p^* = 0.2461$, equivalent to TVaR at p^*. The method `distortion(P, risk, bound)` returns the achieving `Distortion`, and `check(P)` reprices each risk and the total from first principles with it. The largest errors are below 10^{-11}.

```
print(pb.distortion(P, '100 xs 250', 'upper'))
print(pb.check(P)[['err', 'total_err']].abs().max().to_dict())
GT(pb.bitvars(P).droplevel('P'), float_format='.4f')
```

```
bitvar(0, 0.98496; 0.09266)
{'err': 4.2454928461665986e-13, 'total_err': 5.4285465012071654e-12}
```

risk	*bound*	value	p0	p1	w1
100 xs 250	lower	3.8102	0.2457	0.2462	0.9083
	upper	11.8107	0.0000	0.9850	0.0927
50 xs 150	lower	14.0979	0.1202	0.9900	0.0470
	upper	18.4966	0.0000	0.7868	0.2283
Two	lower	158.7969	0.0000	0.9900	0.0909
	upper	158.7969	0.0000	0.9900	0.0909

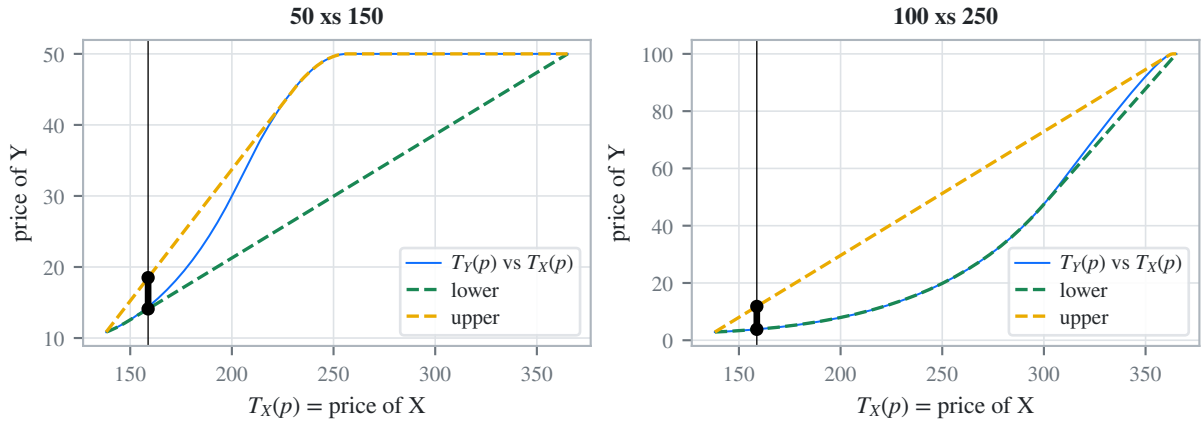
Bounds as the premium varies

The hull does not depend on the premium, so `bounds` accepts a list of premiums at almost no cost. The width of each range is zero at L, where only the identity distortion $g(s) = s$ prices $X \wedge a$ to L, and it grows with the premium. The plot shows the curve of each layer against $\mathrm{TVaR}_p(X)$, its lower and upper hulls, and the slice at P.

```
Ps = [pb.premium_range[0], 145, 150, P, 170, 200]
GT(pb.bounds(Ps).width.unstack(), float_format='.3f')
```

P	100 xs 250	50 xs 150	Two
138.164	0.000	0.000	0
145.000	2.726	1.647	0
150.000	4.670	2.667	0
158.797	8.000	4.399	0
170.000	12.066	6.604	0
200.000	21.720	12.508	0

```
fig, axs = figure('full', cols=2)
pb.plot(items=list(layers), P=P, axs=axs);
```



The curve of 50 xs 150 is flat at 50 for $p \geq \Pr(X < 200) = 0.871$, where $\text{TVaR}_X = 256.6$, because every outcome in the tail then exhausts the layer. At P the curve of 100 xs 250 lies on its lower hull, so the lower bound is TVaR at p^*, and its upper hull is far above the curve.

The uniform reference and the Gini index

With `x_source='uniform'` the reference is the uniform distribution on $[0, 1]$, with $\text{TVaR}_p = (1 + p)/2$. A distortion prices it at $\int_0^1 g(s)\,ds$, the area under g, so a premium P for the uniform fixes the Gini index of g at $2P - 1$, the `gini` column of `distortion_df`. In Kusuoka terms $2P - 1 = \int p\,\mu(dp)$, the mean TVaR level, which `mean_kusuoka_level(P)` returns. The bounds are the range of the price of Y over all distortions with a given Gini index. The table shows the range of the price of the total of `Two` at the Gini index of each calibrated distortion. Each range contains 158.797, the price of the total under every calibrated distortion: `ccoc` is near the top of its range and `tvar` near the bottom.

```
gl = PricingBounds('uniform', {'Two': port}, a=a)
gini = port.distortion_df.gini
GT(pd.DataFrame({k: gl.bounds((1 + g) / 2).iloc[0] for k, g in gini.items()}).T
   .assign(gini=gini)[['gini', 'lower', 'upper', 'width']]
   .rename_axis('distortion'), float_format='.4f')
```

distortion	gini	lower	upper	width
ccoc	0.0909	145.7791	159.0098	13.2307
ph	0.1556	151.1997	173.8484	22.6487
wang	0.1868	153.8115	180.9983	27.1867
dual	0.2196	156.5622	188.5280	31.9659
tvar	0.2461	158.7814	194.6031	35.8217

Refused inputs

A premium outside `premium_range` is refused, and so is a call to `mean_kusuoka_level` when the reference is not the uniform.

```
refuse(lambda: pb.bounds(400), lambda: pb.mean_kusuoka_level(P))
```

```
P=[400.] outside feasible premium range [138.164, 365.125] (= [E[X], TVaR at
s_floor truncation]).
mean_kusuoka_level requires the uniform X-source (pass x_source='uniform').
```

Using profit commissions

Profit commissions on an aggregate quota share with the DecL pc clause

Stephen J. Mildenhall

2026-11-01

A **profit commission** returns to the cedant a share of the reinsurer's profit on a treaty, settled after the year from the treaty's own experience. It is common on quota shares and appears on some excess of loss treaties. With ceded loss A, a fixed ceded premium P_C and ceded loss ratio $\lambda = A/P_C$, a profit share s and an allowance a for the reinsurer's expenses and margin, the commission is

$$K = s(1 - \lambda - a)^+ P_C,$$

which is zero once the loss ratio reaches $1 - a$. As a function of the loss ratio it is a put option struck at $1 - a$: the cedant shares in a good year and pays nothing back in a bad one.

In DecL the clause `pc <share> after <allowance>` decorates one layer of an `aggregate net of` program, and the layer needs a `deposit`, `rol`, or `rate` premium to form the loss ratio. The profit commission is one of the variable features of DecL, with sliding scale commissions, loss corridors, swing rating, and retro rating. A program takes at most one of them, the reinsurance features decorate aggregate covers only, and all take effect in a `pnl` or `xpnl` (see Using pnl and xpnl). The commission is booked in the expense column of the cover, `E` in `economic_ratios_df`, as a random credit to the cedant.

```
aggregate net of 50% po inf xs 0 rate 100% pc 0.35 after 0.1 as QS
aggregate net of 50% po inf xs 0 rate 100% pc 50% after 15% as QS
aggregate net of 2000 xs 8000 rate 6% pc 0.25 after 0.2 as "Stop loss"
```

The variable feature pages share one base book: premium 10,000 at a 70% loss ratio, a policy limit of 2,000, lognormal severity with mean 150 and CV 2.5, gamma mixed claim counts with mixing CV 0.25, and expenses of 25% of premium. The setup cell wraps the book in an `xpnl` with a slot for each reinsurance program.

```
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
book = '''agg Book
  10000 premium at 0.7 lr
  2000 xs 0
  sev lognorm 150 cv 2.5 {occurrence}
  mixed gamma 0.25 {aggregate}'''
program = '''xpnl {name}
  inherit premium
  less
    ''' + book + '''
```

```
  less
    0.25 premium expenses'''
def quota_share(name, terms):
    """The base book under a 50% quota share at a 100% rate with the given terms."""
    return build(program.format(name=name, occurrence='',
                 aggregate=f'aggregate net of 50% po inf xs 0 rate 100% {terms} as QS'))
```

```
1.53.0
```

A profit commission on a 50% quota share

The cover is a 50% quota share of the base book at 100% of the subject premium, so the ceded premium is 5,000 and the ceded loss ratio equals the gross loss ratio, with mean 70%. The reinsurer returns 35% of its profit after an allowance of 10%, a commission rate of $0.35(0.9 - \lambda)^+$: 7% at the expected loss ratio and at most 31.5%, in a year with no loss. The quota share pays no other commission.

```
pc = quota_share('PC', 'pc 0.35 after 0.1')
print(pc.engine.variable_terms)
qd(pc)
```

```
ProfitCommissionTerms(share=0.35, allowance=0.1)
PnL('PC': 2 groups, 6 legs -> 'margin')

                         EX     SD       CV     Skew     P01  Median     P99
Step  Side
Gross Consideration   10000      0        0            10000   10000   10000
      Obligation      -9500   2704 -0.28463 -0.67687  -17071   -9193   -4633
      Margin            500   2704    5.408 -0.67687   -7071     807    5367
QS    Consideration   -5000      0        0            -5000   -5000   -5000
      Obligation     3923.6 1037.9  0.26452   1.2482  2268.2  3750.2  7285.5
      Margin        -1076.4 1037.9 -0.96419   1.2482 -2731.8 -1249.8  2285.5
      Net           -576.43 1679.7  -2.9141 -0.37235 -4785.5 -442.78  2635.2
All   Consideration    5000      0        0             5000    5000    5000
      Obligation    -5576.4 1679.7 -0.30122 -0.37235 -9785.5 -5442.8 -2364.8
      Margin        -576.43 1679.7  -2.9141 -0.37235 -4785.5 -442.78  2635.2
      Impact        -1076.4 1037.9 -0.96419   1.2482 -2731.8 -1249.8  2285.5
```

The `QS` step is the cover from the cedant's side: the ceded premium of 5,000 against the recovery and the profit commission, which together make the obligation row, with mean 3,924. The ledger `economic_df` lists the profit commission as its own leg, with mean 424 and standard deviation 349. It is zero in the 1 in 100 bad year, column `κ01`, and 1,202 in the 1 in 100 good year, `κ99`.

```
GT(pc.economic_df.loc['QS', ['EX', 'SD', 'κ01', 'κ50', 'κ99']], float_format=',.0f')
```

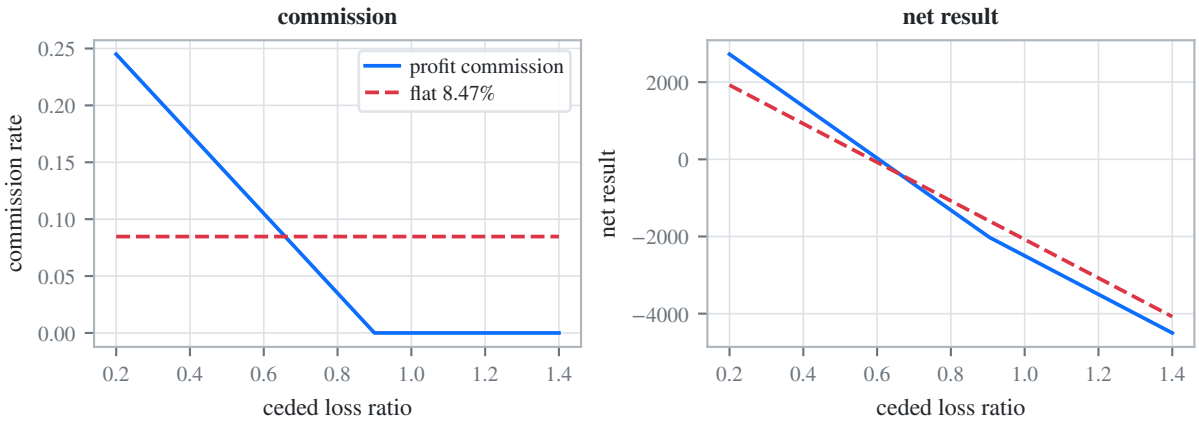
Side	*Label*	EX	SD	κ01	κ50	κ99
Consideration	QS premium	-5,000	0	-5,000	-5,000	-5,000
Obligation	QS recovery	3,500	1,352	7,286	3,346	1,066
	profit commission	424	349	0	404	1,202
	Total	3,924	1,038	7,286	3,750	2,268
Margin	Total	-1,076	1,038	2,286	-1,250	-2,732
	Net	-576	1,680	-4,786	-443	2,635

The mean commission, 424, is 8.47% of the ceded premium, more than the 7% paid at the expected loss ratio. Since $x^+ = x + (-x)^+$, the mean rate is $\mathsf{P}\,0.35(0.9 - \lambda)^+ = 0.07 + 0.35\,\mathsf{P}(\lambda - 0.9)^+$. The commission is floored at zero, so the years with a loss ratio above 90% do not reduce it, and they add 1.47 points to its mean.

Profit commission and net result against a flat commission

To compare shapes rather than prices, the flat cover pays a ceding commission equal to the expected profit commission, 8.47% of the ceded premium. The gross loss is $10{,}000\lambda$ and the recovery $5{,}000\lambda$, so the net result, premium less expense and ceded premium, plus commission and recovery, less gross loss, is $2{,}500 + 5{,}000(c(\lambda) + \lambda) - 10{,}000\lambda$ for a commission rate c. The left panel shows the two commission rates and the right panel the two net results.

```
rate = -pc.economic_ratios_df.loc['QS', 'E'] / 5000
flat = quota_share('Flat', f'cede {rate:.6f}')
lr = np.linspace(0.2, 1.4, 241)
rates = {'profit commission': pc.engine.variable_terms.phi(lr), f'flat {rate:.2%}':
np.full_like(lr, rate)}
fig, (ax0, ax1) = figure('full', cols=2)
for (k, c), ls in zip(rates.items(), ['-', '--']):
    ax0.plot(lr, c, ls=ls, label=k)
    ax1.plot(lr, 10000 - 2500 - 5000 + 5000 * (c + lr) - 10000 * lr, ls=ls, label=k)
ax0.set(xlabel='ceded loss ratio', ylabel='commission rate', title='commission')
ax1.set(xlabel='ceded loss ratio', ylabel='net result', title='net result')
ax0.legend();
```



Below a loss ratio of 90% the commission rate falls 0.35 points per point of loss ratio, so the net result falls at 1.35 times the slope of the flat cover. The two lines cross at a loss ratio of about 66%, below which the profit commission pays more, and above 90% they are parallel, the profit commission 424 below. The table compares the two covers over the whole distribution, with `P01` the 1 in 100 bad result.

```
df = pd.DataFrame({k: [-x.economic_ratios_df.loc['QS', 'E'], x.est_m, x.est_sd, x.q(0.01),
x.cdf(0)]
                   for k, x in {'flat': flat, 'profit commission': pc}.items()},
                  index=['commission', 'mean', 'SD', 'P01', 'Pr(loss)']).T
df.index.name = 'cover'
GT(df, float_format=',.0f', ratio_cols=['Pr(loss)'])
```

cover	commission	mean	SD	P01	Pr(loss)
flat	424	-576	1,352	-4,362	62.9%
profit commission	424	-576	1,680	-4,786	60.0%

The two covers have the same mean result, −576, which is negative because the cedant keeps all of its 2,500 expense against half the premium. The profit commission raises the standard deviation (SD) of the net result from 1,352 to 1,680 and lowers `P01` from −4,362 to −4,786, by the 424 of flat commission it does not pay in a bad year. The probability of a loss, a result of zero or less, falls from 62.9% to 60.0%, since the profit commission pays more than the flat commission in good years.

Refused programs

The profit share must lie between 0 and 1. A program takes at most one variable feature, so a profit commission on the quota share and a corridor on a second aggregate layer, 2000 xs 8000, in the same program are refused. A profit commission replaces the flat commission across its aggregate program, so a `cede` on the same layer or on another aggregate layer is refused too.

```
refuse(lambda: quota_share('R1', 'pc 1.2 after 0.1'),
       lambda: quota_share('R2', 'pc 0.35 after 0.1 and 2000 xs 8000 rate 6% corridor 0.5 po 0.5
       xs 1'),
       lambda: quota_share('R3', 'cede 0.25 pc 0.35 after 0.1'),
       lambda: quota_share('R4', 'pc 0.35 after 0.1 and 2000 xs 8000 rate 6% cede 0.2'))
```

```
ProfitCommissionTerms: share must be in [0, 1], got 1.2.
DecL: at most one variable-rating feature per program (decision 0: no stacking);
found 2.
DecL: 'pc' replaces the fixed 'cede' commission; give one or the other.
DecL: 'pc' replaces the fixed 'cede' commission across its aggregate program, so
a 'cede' on another layer would not be booked; give one or the other.
```

Using recipes

The Underwriter recipe base: the library included with aggregate, the reproductions corpus and your own DecL

Stephen J. Mildenhall

2026-10-30

Every named DecL declaration is stored in a **recipe base**, a map (kind, name) ↦ `Recipe`. A recipe contains the parsed specification, the program, its `note`, `tags` and `hints`, and its source: the file it came from, or `'session'` for a program built in the current session. A later entry under the same key replaces an earlier one. An `Underwriter` holds a recipe base, and the module-level `build` is an underwriter that reads `library.agg`, the recipe library included with aggregate, on first use. A program consisting of a bare name builds the stored entry, and the references `agg.NAME`, `sev.NAME` and `dist.NAME` copy a stored entry into a new program.

The recipe base is held in memory. Nothing is written to disk until a call to `to_agg`, which writes to `~/.aggregate` by default, the folder that `Underwriter(databases='user')` reads. The code on this page writes only to a temporary folder and removes it at the end.

```
agg My.Book 50 claims sev lognorm 10 cv 1 poisson   # stored as (agg, My.Book)
My.Book                                             # builds the stored entry
port My.Port agg.My.Book agg.BasicBook              # agg.NAME copies in an entry
```

The package includes `reproductions.agg`[1], the DecL programs of the *Reproductions* book. They rebuild published actuarial and numerical results, and each is tagged with the paper it comes from. There is an analogous recitations corpus running through textbook exercises.

```
import tempfile
from pathlib import Path
import pandas as pd
from aggregate import build, qd, Underwriter, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

The library recipe

The recipe library included with aggregate, `library.agg`, is read on the first access to the recipe base of `build`. The code below shows how many entries it holds of each kind. The method `discover` filters a recipe base by a regular expression on the name, by `kind` and by `tags`; an entry matches only when it has every tag given.

[1] From aggregate 1.1.0

```
lib = build.recipes
print(len(lib), [p.name for p in build.databases])
wprint(lib.index.get_level_values('kind').value_counts().to_dict())
GT(build.discover(kind='agg', tags='role:intro').head(3))
```

```
206 ['library.agg']
{'agg': 120, 'port': 19, 'sev': 17, 'bvagg': 14, 'xpnl': 14, 'pnl': 9,
'distortion': 7, 'stable': 4, 'chfagg': 1, 'nttagg': 1}
```

name	program
BasicBiasedCoins	agg BasicBiasedCoins dfreq [1000] dsev [0 1] [0.51 0.49]
BasicBook	agg BasicBook 250 claims 1000 xs 0 sev lognorm 100 cv 1.5 poisson
BasicDfreq	agg BasicDfreq dfreq [1 2 3] dsev [1 2 10]

The method `recipe` returns one entry without building it, and its `as_read` attribute is the entry as it appears in its file, with its note and tags. The program `BasicBook`, a bare name, builds the entry.

```
wprint(build.recipe('BasicBook').as_read)
print()
qd(build('BasicBook'))
```

```
agg BasicBook
  250 claims
  1000 xs 0
  sev lognorm 100 cv 1.5
  poisson
  note{A basic realistic insurance book with Poisson frequency, lognormal
  severity and an occurrence limit. This is the workhorse compound
  distribution.}
  tags{topic:aggregate, role:hero, role:intro}

Aggregate object: BasicBook. Frequency distribution poisson. Severity
lognorm, [0, 1,000], bounded; 1,000 xs 0. Updated with bucket size 1 and
log2 = 16. Validation: not unreasonable.

       Mean     SD       CV     Skew   P01 Median   P99
X
Freq    250 15.811 0.063246 0.063246
Sev  98.137 128.91   1.3136   3.5092     4     55   693
Agg   24534 2561.7  0.10441  0.19862 18954  24449 30862
```

The reproductions corpus

The reproductions corpus is not in the default recipe base of `build`, so we read it into an underwriter of our own. An underwriter reads only the files given in its `databases` argument: a file name, a glob, a path, or one of `'default'`, `'user'` and `'all'`. It can be loaded by setting `databases='reproductions'`. Entries are named `<CitationKey>.<name>` and tagged `topic:<CitationKey>`, so a tag selects the programs from one paper.

```
rep = Underwriter(name='Reader', databases='reproductions')
papers = pd.Series([t for ts in rep.recipes.tags for t in ts]).value_counts()
print(len(rep.recipes), len(papers))
wprint(papers.head(5).to_dict())
GT(rep.discover(tags='topic:Gerber1982'))
```

```
143 30
{'topic:Bear1990': 27, 'topic:Mack2003': 17, 'topic:Jin2016': 14,
'topic:Pittarello2024': 13, 'topic:Liu2026': 7}
```

name	program
Gerber1982.Gerber	agg Gerber1982.Gerber 1 claim sev 2 * uniform + 1 poisson
Gerber1982.Gerber.v2	agg Gerber1982.Gerber.v2 10 claims sev 2 * uniform + 1 poisson
Gerber1982.Gerber.v3	agg Gerber1982.Gerber.v3 100 claims sev 2 * uniform + 1 poisson

The `recipes` frame of an underwriter has one row per entry. Its `note` column flags whether a note exists but omits the text, so a search of the notes goes through `recipe`. The program is a column, so `str.contains` searches it directly.

```
notes = pd.Series({n: rep.recipe(n, k).note for k, n in rep.recipes.index})
for name, note in notes[notes.str.contains('Exhibit|stop loss')].items():
    wprint(f'{name:21s} {note}')
progs = rep.recipes.program
wprint(progs[progs.str.contains('splice')].index.get_level_values('name').tolist())
```

```
Bodoff2017.B          @Bodoff2017, Exhibit 4's Pareto severity under a policy
                      limit.
Bodoff2017.B.v2       @Bodoff2017, Exhibit 4's Pareto severity under a policy
                      limit.
Gerber1982.Gerber     @Gerber1982, a compound Poisson on a uniform severity, for
                      the stop loss table.
Gerber1982.Gerber.v2  @Gerber1982, a compound Poisson on a uniform severity, for
                      the stop loss table.
Gerber1982.Gerber.v3  @Gerber1982, a compound Poisson on a uniform severity, for
                      the stop loss table.
Venter1983.Venter     @Venter1983, Exhibit 3, a compound Poisson over a
                      tabulated claim size law.
['Mack2003.Riebesell', 'Mack2003.Riebesell.v2', 'Denuit2025a.Denuit.Pool',
'Denuit2025a.Denuit.Pool.v2', 'Denuit2025a.Denuit.PoolAlt',
'Denuit2025a.Denuit.PoolB']
```

A corpus entry builds by name, like a library entry. `Gerber1982.Gerber` is a compound Poisson with one expected claim and a severity uniform on $[1, 3]$.

```
qd(rep('Gerber1982.Gerber'))
```

```
Aggregate object: Gerber1982.Gerber. Frequency distribution poisson.
Severity uniform, [0, 3], bounded. Updated with bucket size 1/1,024 and
log2 = 18. Validation: not unreasonable.

      Mean      SD      CV   Skew    P01 Median    P99
X
Freq     1       1       1      1
Sev      2 0.57735 0.28868      0 1.0195      2 2.9805
Agg      2  2.0817  1.0408 1.1086      0 1.7188  8.293
```

A scratch recipe base from fork

The method `fork` copies a recipe base in a few microseconds, sharing its parsed entries. What you build in a fork stays in the fork, and the recipe base of `build` never includes it. `load` and `fork` are

thread safe, but concurrent builds on one underwriter are not, so in a program with several threads give each thread its own fork.

```
mine = build.fork('Mine')
mine.build('agg My.Book 50 claims sev lognorm 10 cv 1 poisson '
           'note{a test book} tags{topic:mine}')
mine.build('port My.Port agg.My.Book agg.BasicBook')
print(('agg', 'My.Book') in mine.recipes.index,
      ('agg', 'My.Book') in build.recipes.index)
```

```
True False
```

The entries you build are those whose source is `'session'`. Their `seq` numbers follow those of the library entries, which are numbered from 0. The method `format_agg(pattern)` returns the session entries whose names match `pattern`, as DecL text.

```
display(GT(mine.recipes.query("source == 'session'")[['seq', 'tags', 'source']]))
wprint(mine.format_agg('My.Book'))
```

kind	*name*	seq	tags	source
agg	My.Book	206	('topic:mine',)	session
port	My.Port	207	()	session

```
agg My.Book 50 claims sev lognorm 10 cv 1 poisson note{a test book}
tags{topic:mine}
```

The method `preview` resolves the references in a program without building anything. Each reference is listed with its kind, its name and its source, here one session entry and one library entry.

```
for ref in mine.preview('port Both agg.My.Book agg.BasicBook').resolved:
    print(ref.kind, ref.name, getattr(ref.source, 'name', ref.source))
```

```
agg My.Book session
agg BasicBook library.agg
```

Saving recipes to a file and reading them back

The method `to_agg` writes the entries built in the current session to a `.agg` file. A relative name is written into `~/.aggregate` and an absolute path is used as given, so the absolute path below keeps this page out of your own recipe files. The `spread` layout puts each clause on its own line.

```
tmp = tempfile.TemporaryDirectory()
path = mine.to_agg(Path(tmp.name) / 'mine.agg', layout='spread')
wprint(path.read_text())
```

```
# written by aggregate 1.43.1 on 2026-10-30 12:33:43
# 2 program(s); pattern='.*', kind='all', source='session'
agg My.Book
  50 claims
  sev lognorm 10 cv 1
  poisson
  note{a test book}
  tags{topic:mine}
```

```
port My.Port
  agg My.Book
    50 claims
    sev lognorm 10 cv 1
    poisson
    note{a test book}
    tags{topic:mine}
  agg BasicBook
    250 claims
    1000 xs 0
    sev lognorm 100 cv 1.5
    poisson
    note{A basic realistic insurance book with Poisson frequency, lognormal
    severity and an occurrence limit. This is the workhorse compound
    distribution.}
    tags{topic:aggregate, role:hero, role:intro}
```

A new underwriter given the file as its `databases` reads it back. The reference `agg.BasicBook` in the portfolio was written out in full, so the file is self-contained.

```
back = Underwriter(name='Back', databases=str(path))
print(back.recipes.index.tolist())
qd(back('My.Book'))
```

```
[('agg', 'My.Book'), ('port', 'My.Port')]
Aggregate object: My.Book. Frequency distribution poisson. Severity
lognorm, [0, inf), subexponential right tail. Updated with bucket size
1/32 and log2 = 16. Validation: not unreasonable.

      Mean     SD      CV    Skew    P01 Median    P99
X
Freq    50 7.0711 0.14142 0.14142
Sev     10     10       1  3.9999 1.0312 7.0625 49.062
Agg    500    100     0.2 0.39999 295.31 493.62    762
```

Refused lookups and calls

A lookup of a name that is not in the recipe base raises `RecipeNotFound`, a subclass of `KeyError`. With the corpus loaded beside the library, some names appear under two kinds: the library defines the severity curves as `sev` entries and the corpus wraps each in an `agg`, so a lookup by name alone is ambiguous and needs `kind=`. `to_agg` does not overwrite an existing file unless `mode='w'`, and `Underwriter` accepts keyword arguments only. The last lines remove the temporary folder.

```
both = build.fork('Both')
both.load('reproductions')
refuse(lambda: build.recipe('No.Such.Book'),
       lambda: both.recipe('Mack2003.Lognorm'),
       lambda: mine.to_agg(path), lambda: Underwriter('reproductions'),
       kind=True)
print(both.recipe('Mack2003.Lognorm', kind='sev'))
tmp.cleanup()
print(path.exists())
```

```
RecipeNotFound: no recipe named 'No.Such.Book'
KeyError: 'Mack2003.Lognorm' is ambiguous across kinds (agg, sev); write
agg.Mack2003.Lognorm or sev.Mack2003.Lognorm, or pass kind= to choose. Shipped
library names are unique across kinds.
```

```
FileExistsError: <tmp>\tmpmibbvyvc\mine.agg already exists; pass mode="w" to
overwrite or mode="a" to append.
TypeError: Underwriter.__init__() takes 1 positional argument but 2 were given
Recipe('sev', 'Mack2003.Lognorm')
False
```

Using reinstatements

Paid reinstatements on an occurrence layer, with a ceded premium that varies with the annual recovery

Stephen J. Mildenhall

2026-10-30

An occurrence layer y xs a recovers $R = \sum_i \min((X_i - a)^+, y)$ in a year from the claims $X_1, \dots, X_N$. Property catastrophe and per-risk treaties rarely give that cover without limit. Each loss to the layer uses up part of its limit, and the cedant **reinstates** the limit by paying a **reinstatement premium**, pro rata to the amount reinstated but not to the time left in the year. With base premium B, rate on line $r = B/y$, and m reinstatements priced at multiples $\alpha_1, \dots, \alpha_m$ of r, the recovery and the ceded premium are

$$\min(R, (m+1)y) \qquad \text{and} \qquad B + r \sum_{j=1}^{m} \alpha_j \min\big((R - (j-1)y)^+, y\big).$$

Both are nondecreasing functions of R, so the ceded premium is random and is largest in the years when the recovery is largest.

The `reinstatements` clause decorates one occurrence layer and needs a base premium clause, `rol`, `deposit`, or `rate`, to fix r. The schedule is written in treaty language, with `free` for a multiplier of zero and counts as digits or the words one to five, or as an explicit list of multipliers. The form `no reinstatements` gives one annual limit and no reinstatement premium; omitting the clause leaves the cover free and unlimited. A `pnl` or `xpnl` books the random premium (see Using pnl and xpnl); a plain `agg` has no premium context and ignores the clause with a warning. Reinstatements are the occurrence counterpart of the variable features on aggregate covers: swing rating, the sliding scale, the profit commission, and the loss corridor.

```
occurrence net of 1000 xs 1000 rol 0.45 reinstatements 1 at 100%
1000 xs 1000 rol 0.45 reinstatements 1 free and 2 at 100%   # four annual limits
1000 xs 1000 rol 0.45 reinstatements [0 1 1]                # the same schedule
1000 xs 1000 deposit 450 reinstatements two at 50%
1000 xs 1000 rol 0.45 no reinstatements                     # one annual limit
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

A reinstated layer on the base book

The variable feature pages share one base book: premium 10,000 at a 70% loss ratio, a policy limit of 2,000, lognormal severity with mean 150 and CV 2.5, gamma mixed claim counts with mixing CV 0.25, and expenses of 25% of premium. The layer 1000 xs 1000 is a working layer, with a rate on line of 45% and one reinstatement at 100%. About one claim a year exceeds 1,000, and 0.27 a year reach the policy limit and exhaust the layer.

```
book = '''agg Book
  10000 premium at 0.7 lr
  2000 xs 0
  sev lognorm 150 cv 2.5 {occurrence}
  mixed gamma 0.25 {aggregate}'''
program = '''xpnl {name}
  inherit premium
  less
    ''' + book + '''
  less
    0.25 premium expenses'''
layer = 'occurrence net of 1000 xs 1000 rol 0.45 {} as Layer'
x = build(program.format(name='Reinstated', aggregate='',
                         occurrence=layer.format('reinstatements 1 at 100%')))
qd(x)
e = x.engine
print(f'claims a year over 1,000 {e.n * e.sevs[0].sf(1000):.4f}, '
      f'at 2,000 {e.n * e.sevs[0].sf(1999.99):.4f}')
```

```
PnL('Reinstated': 2 groups, 5 legs -> 'margin')

                         EX     SD       CV     Skew    P01  Median   P99
Step  Side
Gross Consideration   10000      0        0           10000   10000 10000
      Obligation      -9500 2704.6  -0.2847 -0.67669 -17080   -9200 -4640
      Margin            500 2704.6   5.4092 -0.67669  -7080     800  5360
Layer Consideration -635.95 194.51 -0.30587 -0.38086   -900    -549  -450
      Obligation     508.15 602.68    1.186   1.0039      0     220  2000
      Margin        -127.8 427.51  -3.3453   1.5169   -450    -329  1100
      Net            372.2 2450.8   6.5847  -0.6246  -6420     620  4930
All   Consideration  9364.1 194.51 0.020772 -0.38086   9100    9451  9550
      Obligation    -8991.9   2358 -0.26223 -0.64943 -15580   -8740 -4620
      Margin          372.2 2450.8   6.5847  -0.6246  -6420     620  4930
      Impact         -127.8 427.51  -3.3453   1.5169   -450    -329  1100
claims a year over 1,000 0.9925, at 2,000 0.2705
```

The `Layer` step shows the effect of the clause. The ceded premium has mean 635.95 and standard deviation (SD) 194.51. It runs from the base premium of 450 in a year with no recovery to 900 once the first limit is used up and reinstated. The recovery has mean 508.15, and its 99th percentile, 2,000, is the cap of two limits. Net of the layer, the cedant's mean result falls from 500 to 372.20 and its SD from 2,704.60 to 2,450.82.

Ceded premium and recovery under three reinstatement terms

The same layer at the same rate on line is built with no clause, which is free and unlimited cover, with one reinstatement at 100%, and with no reinstatements. The table shows the mean and SD of the ceded premium, the mean and 99th percentile of the recovery, and the mean and SD of the cedant's net result.

```
terms = {'none': '', '1 at 100%': 'reinstatements 1 at 100%',
         'no reinstatements': 'no reinstatements'}
rows = {}
for k, v in terms.items():
    s = build(program.format(name='R', occurrence=layer.format(v),
                             aggregate='')).summary_df
    rows[k] = [-s.loc[('Layer', 'Consideration'), 'EX'],
               s.loc[('Layer', 'Consideration'), 'SD'],
               s.loc[('Layer', 'Obligation'), 'EX'],
               s.loc[('Layer', 'Obligation'), 'P99'],
               s.loc[('All', 'Margin'), 'EX'], s.loc[('All', 'Margin'), 'SD']]
df = pd.DataFrame.from_dict(rows, orient='index', columns=[
    'premium', 'premium SD', 'recovery', 'recovery P99', 'net', 'net SD'])
df.index.name = 'reinstatements'
GT(df, float_format=',.2f')
```

reinstatements	premium	premium SD	recovery	recovery P99	net	net SD
none	450.00	0.00	525.16	2,660	575.16	2,325.77
1 at 100%	635.95	194.51	508.15	2,000	372.20	2,450.82
no reinstatements	450.00	0.00	413.21	1,000	463.21	2,471.50

Without the clause the layer recovers 525.16 on average for a fixed premium of 450, and its recovery exceeds two limits in more than 1% of years. One paid reinstatement caps the recovery at 2,000, which takes 17.01 off the mean recovery, and adds a mean reinstatement premium of 185.95. That premium is the rate on line, 0.45, times 413.21, the mean recovery from the first limit alone, which is the recovery under `no reinstatements`. The reinstatement premium is due in the years with large recoveries, so it offsets part of the protection: the cedant's net SD under one paid reinstatement, 2,450.82, sits between the 2,325.77 of unlimited cover and the 2,471.50 of a single limit.

Refused programs

A `reinstatements` clause needs a base premium on its layer, decorates an occurrence layer only, and requires that layer to be the only occurrence layer.

```
reinst = 'rol 0.45 reinstatements 1 at 100%'
refuse(program.format(name='R1', aggregate='',
                      occurrence='occurrence net of 1000 xs 1000 '
                                 'reinstatements 1 at 100%'),
       program.format(name='R2', occurrence='',
                      aggregate=f'aggregate net of 4000 xs 8000 {reinst}'),
       program.format(name='R3', aggregate='',
                      occurrence='occurrence net of 500 xs 500 rol 0.9 and '
                                 f'1000 xs 1000 {reinst}'))
```

```
DecL: a 'reinstatements' clause needs a base premium clause (deposit / rol /
rate) on the same layer; the base rate on line r = base_premium / limit would
otherwise be undefined ([reins-premium]).
DecL: a 'reinstatements' clause decorates an occurrence layer, not aggregate
reinsurance.
DecL: 'reinstatements' require a single occurrence layer; found 2
([reins-single-layer]). A reinstated layer cannot share the occurrence tier with
other layers (the engine ceiling is the 2-D (L, R) joint).
```

Using renewal counts

Renewal claim counts with a years exposure and the wait and dwait clauses

Stephen J. Mildenhall

2026-10-30

A Poisson claim count is the count of a process whose waiting times between claims are exponential. A renewal count keeps the independence and drops the exponential: the waits $W_1, W_2, \ldots$ are independent and identically distributed with mean μ, variance σ^2, and third central moment μ_3. With $S_k = W_1 + \cdots + W_k$ the count over $[0, T]$ is $N(T) = \max\{k : S_k \le T\}$, so $\{N(T) \ge k\} = \{S_k \le T\}$ and $\Pr(N(T) = k) = F^{*k}(T) - F^{*(k+1)}(T)$, where F is the wait distribution. The renewal count is the claim count of the Sparre-Andersen risk model, and the method `Aggregate.wiener_hopf` computes its probability of eventual ruin, as described in Using pollaczeck_khinchine and wiener_hopf.

For a nonlattice wait law with a finite third moment, renewal theory gives

$$\mathsf{P}N(T) = \frac{T}{\mu} + \frac{\sigma^2 - \mu^2}{2\mu^2} + o(1), \qquad \operatorname{Var} N(T) = \frac{\sigma^2 T}{\mu^3} + \frac{1}{12} + \frac{5\sigma^4}{4\mu^4} - \frac{2\mu_3}{3\mu^3} + o(1),$$

so the index of dispersion $\operatorname{Var} N / \mathsf{P}N$ tends to σ^2/μ^2, the squared coefficient of variation (CV) of the wait. Regular waits, with CV below 1, give an underdispersed count, and bursty waits, with CV above 1, an overdispersed one. The library computes the count probabilities from the convolution powers F^{*k} by FFT, not from these expansions.

A `T years` exposure[1] paired with a `wait` or `dwait` clause in place of the frequency gives a renewal count. The clause `wait` accepts any severity specification, applied on the time axis, including a layer such as `wait 2 xs 1 expon`. The clause `dwait` takes outcomes and probabilities like `dsev`, and a trailing `!` allows a defective, or terminating, wait law whose probabilities sum to less than 1. The clause `at r rate` sets a premium of Tr for information; the count depends on the wait law alone.

```
agg R 10 years sev lognorm 100 cv 1 wait expon               # 10 claims ... poisson
agg R 10 years dsev [1] wait gamma 1 cv 0.5                  # regular arrivals
agg R 10 years dsev [1] wait [0.5 2] * expon wts [2/3 1/3]   # bursty arrivals
agg R 3 years dsev [1] dwait [1]                             # exactly 3 claims
agg R 2 years dsev [1] dwait [1 2] [0.4 0.5] !               # terminating
agg R 5 years at 300 rate sev lognorm 100 cv 1 wait 2 xs 1 expon   # layered wait
```

```
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.0

Exponential waits and the Poisson count

Exponential waits with mean one over ten years give the Poisson count with mean ten. The attribute `Frequency.freq_df` lists the count probabilities `p` beside the Poisson probabilities with the same mean, `po_p`, and the two agree to within 10^{-8}.

```
a = build('agg Ren 10 years dsev [1] wait expon')
qd(a)
df = a.frequency.freq_df
print(f'max |p - po_p| = {(df.p - df.po_p).abs().max():.1e}')
```

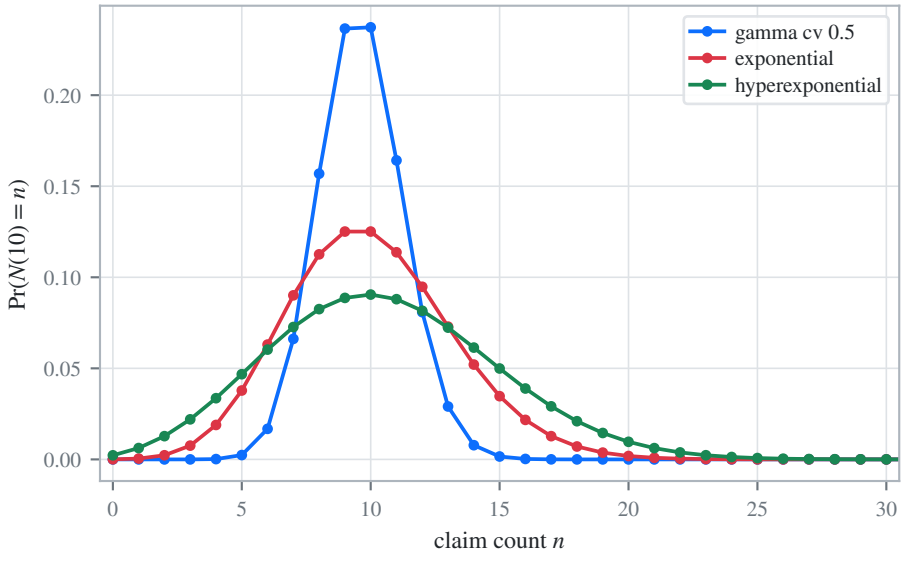
```
Aggregate object: Ren. Frequency distribution renewal. Severity
dhistogram, [1, 1], bounded; atoms [1]. Updated with bucket size 1 and
log2 = 6. Validation: not unreasonable.

      Mean     SD      CV     Skew P01 Median P99
X
Freq    10 3.1623 0.31623 0.31623
Sev      1      0       0          1      1   1
Agg     10 3.1623 0.31623 0.31623   3     10  18
max |p - po_p| = 6.3e-09
```

Renewal counts for regular and bursty arrivals

The figure compares the renewal count over ten years for three wait laws with mean one: a gamma with CV 1/2, the exponential, and a hyperexponential mixture of exponentials with means 1/2 and 2 and CV $\sqrt{2}$. The mean counts are close, but the spreads differ widely.

```
waits = {'gamma cv 0.5': 'gamma 1 cv 0.5', 'exponential': 'expon',
         'hyperexponential': '[0.5 2] * expon wts [2/3 1/3]'}
books = {k: build(f'agg Ren 10 years dsev [1] wait {w}') for k, w in waits.items()}
fig, ax = figure('three-quarter')
for k, b in books.items():
    p = b.frequency.freq_df.p
    ax.plot(p.index, p, 'o-', ms=3, label=k)
ax.set(xlim=[-0.5, 30.5], xlabel='claim count $n$', ylabel=r'$\Pr(N(10)=n)$')
ax.legend();
```



Renewal count moments and their asymptotic expansions

The table compares the computed mean and variance of the renewal count over ten years with the asymptotic expansions, for gamma, exponential, and hyperexponential waits with mean one. The wait moments are taken from the wait law stored in the frequency: `wait_components` lists its `Severity` objects and `wait_weights` their mixture weights. At $T = 10$ the asymptotic mean and variance match the computed moments to five figures or better.

```
def asymptotics(b):
    f = b.frequency
    w = np.atleast_1d(f.wait_weights)
    m1, m2, m3 = w @ np.array([sev.moms() for sev, *_ in f.wait_components])
    s2, c3, T = m2 - m1 ** 2, m3 - 3 * m1 * m2 + 2 * m1 ** 3, f.years
    return {'wait cv': s2 ** 0.5 / m1, 'mean': b.est_m,
            'asymptotic mean': T / m1 + (s2 - m1 ** 2) / (2 * m1 ** 2),
            'variance': b.est_var,
            'asymptotic variance': s2 * T / m1 ** 3 + 1 / 12
            + 5 * s2 ** 2 / (4 * m1 ** 4) - 2 * c3 / (3 * m1 ** 3)}

df = pd.DataFrame({k: asymptotics(b) for k, b in books.items()}).T
GT(df.rename_axis('wait law'), float_format='.6f')
```

wait law	wait cv	mean	asymptotic mean	variance	asymptotic variance
gamma cv 0.5	0.500000	9.625000	9.625000	2.578125	2.578125
exponential	1.000000	10.000000	10.000000	10.000000	10.000000
hyperexponential	1.414214	10.499977	10.500000	18.750296	18.750000

Discrete waits

The `dwait` clause takes outcomes and probabilities, like `dsev`. Unit waits over three years give exactly three claims, since the arrival on the horizon counts, and the count has CV zero. A zero wait is a simultaneous claim, so `dwait [0 1] [0.5 0.5]` produces clusters of claims, and over two years the mean count is 5. Probabilities summing to 0.9 with `!` leave a defect of 0.1: each wait is infinite with probability 0.1, and no claim arrives after the first such wait.

```
for program in ['agg Det 3 years dsev [1] dwait [1]',
                'agg Clu 2 years dsev [1] dwait [0 1] [0.5 0.5]',
                'agg Ter 2 years dsev [1] dwait [1 2] [0.4 0.5] !']:
    b = build(program)
    print(f'{program:48s} mean {b.est_m:6.4f}  cv {b.est_cv:6.4f}  '
          f'defect {b.frequency.wait_defect:.1f}')
```

```
agg Det 3 years dsev [1] dwait [1]               mean 3.0000  cv 0.0000  defect 0.0
agg Clu 2 years dsev [1] dwait [0 1] [0.5 0.5]   mean 5.0000  cv 0.4899  defect 0.0
agg Ter 2 years dsev [1] dwait [1 2] [0.4 0.5] ! mean 1.0600  cv 0.4777  defect 0.1
```

An aggregate with a renewal count

A renewal count combines with any severity X, as a Poisson count does. The aggregate variance is $\mathsf{P}N \operatorname{Var}X + \operatorname{Var}N\,(\mathsf{P}X)^2$, so regular arrivals narrow the aggregate. The example has gamma waits with mean 1/2 and CV 1/2 over five years, an expected 9.625 claims, compared with a Poisson count

with the same mean on the same grid, and `at 300 rate` sets a premium of 1,500. The renewal count has CV 0.17 against 0.32 for the Poisson, and the aggregate CV is 0.36 against 0.46.

```
r = build('agg Ren 5 years at 300 rate sev lognorm 100 cv 1 wait gamma 0.5 cv 0.5')
p = build(f'agg Po {r.n:.4f} claims sev lognorm 100 cv 1 poisson', bs=r.bs, log2=r.log2)
qd(r)
qd(p)
print(r.pprogram)
print(f'premium {r.exp_premium:,.0f}')
```

```
Aggregate object: Ren. Frequency distribution renewal. Severity lognorm,
[0, inf), subexponential right tail. Updated with bucket size 1/4 and
log2 = 16. Validation: not unreasonable.

      Mean     SD      CV    Skew    P01 Median   P99
X
Freq 9.625 1.6057 0.16682 0.15098
Sev    100    100       1  3.9998  10.25  70.75 490.5
Agg  962.5 349.33 0.36294  1.0992 373.75 911.25  2029
Aggregate object: Po. Frequency distribution poisson. Severity lognorm,
[0, inf), subexponential right tail. Updated with bucket size 1/4 and
log2 = 16. Validation: not unreasonable.

      Mean     SD      CV    Skew   P01 Median    P99
X
Freq 9.625 3.1024 0.32233 0.32233
Sev    100    100       1  3.9998 10.25  70.75  490.5
Agg  962.5 438.75 0.45584 0.91166   208 903.25 2256.5
agg Ren
  5 years at 300 rate
  sev lognorm 100 cv 1
  wait gamma 0.5 cv 0.5
premium 1,500
```

The renewal aggregate has the slightly higher median, 911 against 903. Its upper quantiles are lower, and at the 99.9% level it is 2,701 against 2,967 for the Poisson.

```
ps = [0.5, 0.9, 0.99, 0.999]
df = pd.DataFrame({'renewal': [r.q(x) for x in ps], 'poisson': [p.q(x) for x in ps]},
                  index=pd.Index([f'{x:g}' for x in ps], name='p'))
GT(df, float_format=',.1f')
```

p	renewal	poisson
0.5	911.2	903.2
0.9	1,415.2	1,540.5
0.99	2,029.0	2,256.5
0.999	2,700.8	2,967.0

Refused programs

The `years` exposure and the `wait` or `dwait` clause are paired by the grammar, so either one without the other is a parse error. The horizon must be a single number, not a vector.

```
refuse('agg X 10 years dsev [1] poisson',
       'agg X 10 claims dsev [1] wait expon',
       'agg X [5 10] years dsev [1] wait expon')
```

```
DecL parse error at line 1, column 27: Unexpected 'poisson'.
DecL parse error at line 1, column 28: Unexpected 'wait'.
DecL: years in a years (renewal) exposure must be a scalar, not a vector
```

Using retro rating

Retrospectively rated premium for a large account, set by the account's own loss between a minimum and a maximum

Stephen J. Mildenhall

2026-10-30

Retrospective rating sets an insured's premium for a policy year after the year ends, from the account's own losses. With account loss L, a basic premium b, a loss conversion factor m, and a minimum ℓ and maximum u, the retrospective premium is

$$P(L) = \min\big(\max(b + mL,\ \ell),\ u\big).$$

Retro plans are written for large commercial accounts, in workers compensation and general and auto liability, whose own experience is credible. The basic premium covers expenses and the charge for the collar. The loss conversion factor loads the losses for claim adjustment. Between the minimum and the maximum the insured pays m for each unit of its own loss, so it keeps most of its risk and buys protection only above the maximum. Swing rating applies the same collared map to the ceded premium of a reinsurance cover; retro applies it to the gross premium of the account.

The clause `retro basic <b> lcm <m> [min <lo>] [max <hi>] premium` is the premium head of a `pnl`, in place of a fixed or inherited premium (see Using pnl and xpnl). The minimum defaults to the basic and the maximum to no cap. The account must have no reinsurance, so that the account loss is the gross loss. An `xpnl` is refused, since there is no cession to walk through. The other variable features apply to reinsurance: reinstatements, the sliding scale, the profit commission, and the loss corridor.

```
pnl NAME retro basic 250 lcm 1.1 min 600 max 1500 premium less <aggregate> [less <expenses>]
retro basic 250 lcm 1.1 max 1500 premium        # minimum 250, the basic
```

```
import numpy as np
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
1.43.1
```

A retro-rated large account

The example is a large casualty account with a guaranteed cost premium of 1,000 at a 65% expected loss ratio, losses limited to 250 per claim, lognormal severity with mean 25 and CV 3, gamma mixed claim counts with mixing CV 0.2, and fixed expenses of 200. The retro plan has basic premium 250, loss conversion factor 1.1, minimum 600, and maximum 1,500. The account is written once, with a placeholder for the premium head.

```
account = '''pnl {name}
  {head}
  less
    agg Account
      1000 premium at 0.65 lr
      250 xs 0
      sev lognorm 25 cv 3
      mixed gamma 0.2
  less
    200 fixed expenses'''
r = build(account.format(
    name='Retro', head='retro basic 250 lcm 1.1 min 600 max 1500 premium'))
qd(r)
loss = r.engine
print(f'loss mean {loss.est_m:.2f}, median {loss.q(0.5):.2f}')
print(f'premium at the minimum {loss.cdf(350 / 1.1):.4f}, '
      f'at the maximum {loss.sf(1250 / 1.1):.4f}')
```

```
PnL('Retro': role=sell, 1 consideration, 2 obligation -> 'margin')

                  EX     SD       CV     Skew     P01  Median  P99
Side
Consideration 962.01 268.94  0.27956  0.44463     600  926.78 1500
Obligation      -850 280.42 -0.32991 -0.73085 -1643.6 -815.25 -364
Margin        112.01 57.161  0.51033  -5.5019 -143.62   113.5  236
loss mean 650.00, median 615.25
premium at the minimum 0.1002, at the maximum 0.0577
```

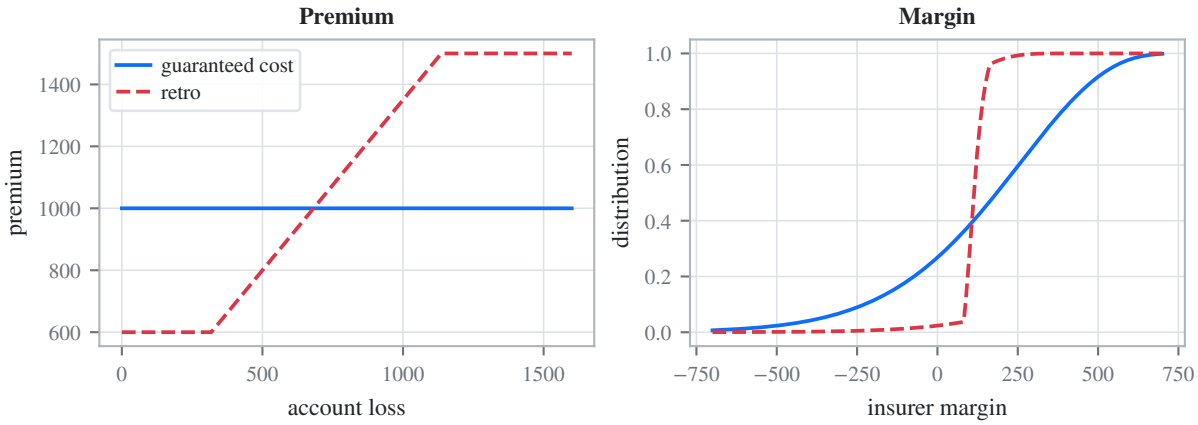
The `Consideration` row is the retrospective premium, with mean 962.01 and standard deviation (SD) 268.94, against an account loss with mean 650 and SD 280.42 (the obligation adds the expense of 200 to the loss). The premium is at the minimum of 600 when the loss is below $350/1.1 = 318.2$, in 10.0% of years, and at the maximum of 1,500 when the loss exceeds $1{,}250/1.1 = 1{,}136.4$, in 5.8% of years. In between it moves with the loss: the median premium, 926.78, is $250 + 1.1 \times 615.25$, where 615.25 is the median loss.

Retro rating against guaranteed cost

The guaranteed cost policy charges the fixed premium of 1,000, the premium of the account (`inherit premium`). The left panel shows the two premiums against the account loss, and the right panel the distribution of the insurer's margin, premium less loss and expense, under each plan.

```
gc = build(account.format(name='GC', head='inherit premium'))
x = np.linspace(0, 1600, 801)
m = np.linspace(-700, 700, 1401)
fig, (ax0, ax1) = figure('full', cols=2)
ax0.plot(x, np.full_like(x, 1000), label='guaranteed cost')
ax0.plot(x, loss.variable_terms.phi(x), ls='--', label='retro')
ax1.plot(m, gc.cdf(m), label='guaranteed cost')
ax1.plot(m, r.cdf(m), ls='--', label='retro')
ax0.set(xlabel='account loss', ylabel='premium', title='Premium')
ax1.set(xlabel='insurer margin', ylabel='distribution', title='Margin')
ax0.legend()
for k, p in {'guaranteed cost': gc, 'retro': r}.items():
    print(f'{k:16s} margin {p.est_m:7.2f}  SD {p.est_sd:6.2f}  '
          f'P01 {p.q(0.01):8.2f}')
```

```
guaranteed cost  margin  150.00  SD 280.42  P01  -643.62
retro            margin  112.01  SD  57.16  P01  -143.62
```



The collar shares the loss back with the insured. Between the two breakpoints the insurer's margin is $250 - 200 + 0.1L$, rising from 81.8 to 163.6, because each unit of loss returns 1.1 in premium. Below a loss of 318.2 the premium stays at 600 and the insurer keeps the savings, with margin $400 - L$; above 1,136.4 the premium stays at 1,500 and the insurer bears the whole loss, with margin $1{,}300 - L$. Retro gives up 37.99 of the insurer's expected margin. In exchange the SD of the margin falls from 280.42 to 57.16 and the 1 in 100 margin improves from −643.62 to −143.62.

Refused programs

Retro is refused in an `xpnl` and on an account with reinsurance.

```
head = 'retro basic 250 lcm 1.1 min 600 max 1500 premium'
refuse(account.format(name='R1', head=head).replace('pnl', 'xpnl', 1),
       account.format(name='R2', head=head).replace(
           'mixed gamma 0.2', 'mixed gamma 0.2 aggregate net of 500 xs 1000'))
```

```
R1: 'xpnl' over a retro program is not supported -- retro has no cession to walk
through. Use 'pnl'.
R2: 'retro' with reinsurance (or another variable feature) is not yet supported;
retro currently requires a book with no inuring reinsurance (the 1-D
net-account-loss = gross-loss case).
```

Using severity parameters

Shape parameters, scale and location, the mean and CV form, and the _ placeholder in a mixture of severity families

Stephen J. Mildenhall

2026-11-01

A continuous severity $X = cZ + b$ is a standard law Z with a scale c and a shift b. The law of Z depends only on its family and its **shape parameters**. A normal, exponential, or uniform Z has no shape parameter, a gamma, lognormal, or Pareto has one, and a beta or Burr has two.

In DecL the numbers after the family word are the shape parameters, in the order `scipy.stats` lists them. Scale and shift are arithmetic around the law, `c * family shapes + b`. A second form, `family mean cv v`, declares a mean and coefficient of variation (CV) and solves for the rest. These forms are used to declare a continuous `scipy.stats` family with up to two shape parameters as a severity, and to mix different families in one severity, where the placeholder `_`[1] holds the place of a family with no shape to give.

```
agg A 10 claims sev 30 * norm + 100 poisson                # normal, mean 100, sd 30
agg B 10 claims sev 25 * gamma 4 poisson                   # gamma, shape 4, scale 25
agg C 10 claims sev gamma 100 cv 0.5 poisson               # gamma, mean 100, CV 0.5
agg D 10 claims sev norm 100 cv 0.3 poisson                # the same law as A
agg E 10 claims sev [100 30] * [lognorm norm] [0.5 _] + [0 100] wts [0.6 0.4] poisson
```

```
import pandas as pd
from aggregate import build, qd, Severity, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.60.0
```

A helper tabulates the mean, standard deviation, and CV of the frozen `scipy.stats` law behind a severity clause, one row for each family in the clause. The frozen law is the attribute `fz` of a severity.

```
def laws(*clauses):
    rows = {}
    for clause in clauses:
        sevs = build(f'agg T 1 claim sev {clause} fixed', update=False).sevs
        for sev in sevs:
            m, v = map(float, sev.fz.stats(moments='mv'))
            sd = v ** 0.5
            label = clause if len(sevs) == 1 else f'{clause}: {sev.sev_name}'
            rows[label] = [m, sd, sd / m if m else float('nan')]
    return pd.DataFrame.from_dict(rows, orient='index', columns=['mean', 'sd', 'cv']
                                  ).rename_axis('clause')
```

[1] From aggregate 1.51.0

Shape parameters, scale, and shift of a severity

The next table shows laws with none, one, and two shape parameters, scaled and shifted by arithmetic. A multiplier scales the mean and the standard deviation, and a shift moves the mean alone.

```
GT(laws('norm', '30 * norm + 100', '5 * expon', '10 * uniform', 'gamma 4',
        '25 * gamma 4', '25 * gamma 4 + 10', '100 * beta 2 3'))
```

clause	mean	sd	cv
norm	0	1.0000	—
30 * norm + 100	100	30.0000	0.3000
5 * expon	5	5.0000	1.0000
10 * uniform	5	2.8868	0.5774
gamma 4	4	2.0000	0.5000
25 * gamma 4	100	50.0000	0.5000
25 * gamma 4 + 10	110	50.0000	0.4545
100 * beta 2 3	40	20.0000	0.5000

The mean and CV form of a severity

The form `family mean cv v` sets the mean μ and CV υ directly. For a family with one shape parameter, the shape is solved from the CV and then the scale from the mean. The two shapes of a beta are solved the same way, on the interval from 0 to the scale. A family with no shape parameter, whose standard law has mean m_0 and standard deviation s_0, takes scale $\upsilon\mu/s_0$ and location $\mu - \text{scale} \cdot m_0$.

The table shows the declared values for each kind of family. A multiplier in front scales the declared mean, and a shift is added afterwards. An exponential with CV 1 needs no location, and with CV 0.5 it must be shifted by half its mean.

```
GT(laws('gamma 100 cv 0.5', '100 * beta 0.5 cv 0.2', 'norm 100 cv 0.3', '2 * norm 100 cv 0.3',
        'expon 10 cv 1', 'expon 10 cv 0.5', 'expon 10 cv 0.5 + 3', 'uniform 10 cv 0.2'))
```

clause	mean	sd	cv
gamma 100 cv 0.5	100	50.0000	0.5000
100 * beta 0.5 cv 0.2	50	10.0000	0.2000
norm 100 cv 0.3	100	30.0000	0.3000
2 * norm 100 cv 0.3	200	60.0000	0.3000
expon 10 cv 1	10	10.0000	1.0000
expon 10 cv 0.5	10	5.0000	0.5000
expon 10 cv 0.5 + 3	13	5.0000	0.3846
uniform 10 cv 0.2	10	2.0000	0.2000

The declared moments are those of the law before it is applied to losses. A normal or a widely spread shifted law can reach below zero, where `sev` clamps it to zero and `ssev` keeps it, as described in Using `ssev`. The normal with mean 100 and CV 1 has 16% of its mass below zero, so under `sev` the severity has a higher mean and a lower CV than were declared.

```
rows = {}
for keyword in ['sev', 'ssev']:
    s = build(f'agg T 1 claim {keyword} norm 100 cv 1 fixed', update=False).sevs[0]
    rows[keyword] = [s.actual_m, s.actual_sd, s.actual_cv]
GT(pd.DataFrame.from_dict(rows, orient='index', columns=['mean', 'sd', 'cv']
                          ).rename_axis('keyword'))
```

keyword	mean	sd	cv
sev	108.3315	86.6653	0.8000
ssev	100.0000	100.0000	1.0000

Mixtures of different severity families

A vector of families shares one list of numbers, and entry i of the list goes to family i. The placeholder _ holds the place of a family with no shape to give. In a mixture of a lognormal and a normal the normal has no shape, so the list is `[0.5 _]`. The scales and shifts are vectors in the same order. The next severity mixes a lognormal with $\sigma = 0.5$ scaled by 100 and a normal with mean 100 and standard deviation 30, and the table shows the law of each family.

```
mix = '[100 30] * [lognorm norm] [0.5 _] + [0 100] wts [0.6 0.4]'
GT(laws(mix))
```

clause	mean	sd	cv
[100 30] * [lognorm norm] [0.5 _] + [0 100] wts [0.6 0.4]: lognorm	113.3148	60.3901	0.5329
[100 30] * [lognorm norm] [0.5 _] + [0 100] wts [0.6 0.4]: norm	100.0000	30.0000	0.3000

A family with two shape parameters needs a second list, and _ holds a place there too. In a mixture of a lognormal and a beta, the lognormal's one shape goes in the first list, and _ holds its place in the second. The table shows the two laws.

```
GT(laws('[lognorm beta] [0.5 2] [_ 3] wts=2'))
```

clause	mean	sd	cv
[lognorm beta] [0.5 2] [_ 3] wts=2: lognorm	1.1331	0.6039	0.5329
[lognorm beta] [0.5 2] [_ 3] wts=2: beta	0.4000	0.2000	0.5000

The mixture is an ordinary severity for an aggregate, and the function `qd` displays the statistics of the aggregate.

```
qd(build(f'agg Mix 10 claims sev {mix} poisson'))
```

```
Aggregate object: Mix. Frequency distribution poisson. Mixed severity
with 2 components. Updated with bucket size 1/8 and log2 = 16.
Validation: not unreasonable.

       Mean      SD       CV     Skew     P01 Median     P99
```

```
X
Freq     10 3.1623 0.31623 0.31623
Sev  107.99 50.896  0.4713  1.9534     31    100 289.75
Agg  1079.9 377.52 0.34959  0.4379 329.38 1052.4 2074.9
```

Refused severity clauses

Numbers a family cannot take are refused and never dropped, in a `sev` clause as in a `chfagg` statement, and the error message gives the form to use where there is one. A placeholder must leave each family with the shapes it takes. A mean and CV the family cannot reach are refused too, and so is a family that does not exist, with the closest names. The class `Severity` refuses the same mistakes where the values show them.

```
refuse(*[f'agg T 1 claim sev {clause} fixed' for clause in [
    'norm 100 30', 'expon 5', 'gamma 4 25', 'beta 2', 'gamma',
    'burr 100 cv 0.5', 'cauchy 10 cv 1',
    '[lognorm norm] [0.5 17]', '[expon pareto] [1 3]', '[lognorm lognorm] [0.5 _]',
    'norm -100 cv 0.3', 'beta 2 cv 0.2', 'lognorml 0.5']],
    lambda: Severity('gamma', sev_a=4, sev_b=25),
    lambda: Severity('norm', sev_a=100))
```

```
sev: norm has no shape parameter; for location 100 and scale 30 write '30 * norm
+ 100', or 'norm 100 cv 0.3'.
sev: expon has no shape parameter; for scale 5 write '5 * expon', for a shift
'expon + 5'.
sev: gamma takes one shape parameter; for shape 4 and scale 25 write '25 * gamma
4', and if 25 is a cv, 'gamma 4 cv 25'.
sev: beta takes two shape parameters, got one. Or give a mean and cv: beta m cv
v.
sev: gamma takes one shape parameter, got none. Or give a mean and cv: gamma m
cv v.
sev: burr has no mean cv form; give its two shapes.
sev: cauchy has no finite mean, so no mean cv form.
sev: norm has no shape parameter, and in a vector of families '_' holds its
place: write '[lognorm norm] [0.5 _]'.
sev: expon has no shape parameter, and in a vector of families '_' holds its
place: write '[expon pareto] [_ 3]'.
sev: lognorm takes one shape parameter, got none in this vector of families.
norm: a mean and cv form needs both positive, got mean -100.0 and cv 0.3.
Otherwise give a scale and shift: c * norm + b.
beta has no law with mean 2.0 and cv 0.2 on [0, 1.0]: the mean over the scale,
m, must be below 1 and the cv squared below (1 - m) / m.
DecL parse error at line 1, column 19: Unknown severity family 'lognorml'. Did
you mean: lognorm, loguniform, powerlognorm?
gamma takes one shape parameter, got sev_b=25; use sev_scale.
norm has no shape parameter, got sev_a=100; use sev_scale and sev_loc.
```

Using sharpen

Auditing the FFT grid with sharpen on Aggregate and Portfolio

Stephen J. Mildenhall

2026-10-30

The bucket sizer selects the bucket size b (`bs`) and the number of buckets $n = 2^{\ell}$, with $\ell =$ `log2`, from the analytic moments, before any FFT runs (Using FFT windows). The method `sharpen()` audits that choice afterward. It updates the object on neighboring grids and scores each by how well the computed law reproduces six analytic moments: the severity and aggregate mean, coefficient of variation (CV), and skewness. Each relative error e_i is divided by its validation tolerance τ_i (ϵ for a mean, 10ϵ for a CV, 100ϵ for a skewness, with $\epsilon =$ `validation_eps`), so a term above 1 fails validation. A moment that is zero or infinite in theory is skipped, and the score is the power mean of the m remaining terms,

$$s = \Big(\frac{1}{m}\sum_{i=1}^{m}\big(|e_i|/\tau_i\big)^2\Big)^{1/2}.$$

The window width is $W = b\,2^{\ell}$. A larger bucket gives a wider window and coarser resolution, so at fixed ℓ the score has one trough in b, and the probe is a line search: double b until the score stops improving, then halve it, at ℓ and $\ell - 1$, and at $\ell + 1$ only when nothing smaller meets the target (default 0.5). Each pair (b, ℓ) tried is a **cell**. A cell is *defective* when its law loses mass off the top of the window, $1 - \sum_k p_k > 10^{-12}$, and is never chosen while a sound cell exists. The winner is the best sound score among the cells that do not grow ℓ. A grid that already meets the target and holds its mass is not probed at all.

The method `sharpen`[1] is available on `Aggregate` and `Portfolio`. The probe records its outcome in the object's own program, so `build(a.program)` rebuilds the sharpened object. A moved grid becomes a `hints` clause, and a confirmed grid, or one whose recommendation was not executed, a `note`.

```
agg Cat 2 claims 1000 xs 0 sev lognorm 10 cv 10 poisson hints{log2=16; bs=1/16}
agg Fine 10 claims sev gamma 50 cv 0.5 poisson note{sharpen: grid confirmed, no change}
```

```
from aggregate import build, exhibits, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.0

A grid that meets the target

The aggregate `Fine`, ten claims from a gamma severity, is sized well by default. Its score, 0.000113, is far under the target of 0.5 and nothing is lost off the top of the window, so no probe runs and the program gains a confirmation note.

```
f = build('agg Fine 10 claims sev gamma 50 cv 0.5 poisson')
f.sharpen()
wprint(f.sharpen_description)
wprint(f.program)
```

```
Sharpen: not run, the grid already scores 0.000113 against a target of 0.5.
agg Fine 10 claims sev gamma 50 cv 0.5 poisson note{sharpen: grid confirmed, no
change}
```

The string `sharpen_explanation` gives the same outcome in prose, together with the definition of the score. Its units are the validation tolerances, so a score of 1 is the boundary between passing and failing validation.

```
wprint(f.sharpen_explanation)
```

```
The sharpen score measures how well the realized grid reproduces the analytic
moments: severity and aggregate mean, CV and skewness, each relative error
divided by its own validation tolerance and combined in a power-2 norm. The
units are tolerance, so a score of 1 sits exactly on the validation pass
boundary and smaller is better. The target here is 0.5. The current grid scores
0.000113, at or under the target, and holds all of its mass, so no probe was run
and nothing was changed. Pass good_enough=0 to force the probe regardless.
```

A grid moved to a finer bucket

The aggregate `Cat` has two claims from a lognormal severity with CV 10, limited to 1000. The severity is bounded, so the sizer takes the limit times a high quantile of the claim count, about 12,000, as the window, and pads it to 16,384 at $b = 1/4$. That bucket is too coarse for a severity with a median near 1, and the default grid fails validation.

```
c = build('agg Cat 2 claims 1000 xs 0 sev lognorm 10 cv 10 poisson')
wprint(c.bs_description)
print(c.validation_description, c.validation_score)
```

```
bounded_small grid: bs=1/4, log2=16, window [0, 16,384] of width 16,384
fails sev mean, agg mean 2.8534000548938114
```

The probe moves the bucket from 1/4 to 1/16 at the same ℓ, and the score from 2.85 to 0.206. The aggregate then passes validation.

```
c.sharpen()
wprint(c.sharpen_description)
print(c.validation_description)
```

```
Sharpen: 11 cells in 0.30s, best score 0.206 vs 2.85 at the center, target 0.5.
1 better-scoring cell rejected as defective (mass off the end of the grid).
Moved: bs 1/4 to 1/16, at log2 16.
not unreasonable
```

The table `sharpen_df` has one row per cell, indexed by the steps `d_bs` and `d_log2` from the starting grid, and its `extent` column is the window width W. At $b = 1/32$ the score is better still, 0.048, but the window of width 2,048 loses 9.4×10^{-8} off the top, so the cell is defective and passed over.

```
cols = ['bs', 'log2', 'extent', 'score', 'deficit', 'defective', 'selected']
GT(c.sharpen_df[cols], formatters={'bs': '.4g', 'extent': ',.0f',
                                   'score': '.3f', 'deficit': '.1e'})
```

d_bs	*d_log2*	bs	log2	extent	score	deficit	defective	selected
-4	0	0.01562	16	1,024	4.004	2.2e-04	True	False
-3	-1	0.03125	15	1,024	4.027	2.2e-04	True	False
	0	0.03125	16	2,048	0.048	9.4e-08	True	False
-2	-1	0.0625	15	2,048	0.208	9.4e-08	True	False
	0	0.0625	16	4,096	0.206	1.1e-14	False	True
-1	-1	0.125	15	4,096	0.813	1.1e-14	False	False
	0	0.125	16	8,192	0.813	1.1e-16	False	False
0	-1	0.25	15	8,192	2.853	-4.4e-16	False	False
	0	0.25	16	16,384	2.853	-4.4e-16	False	False
1	-1	0.5	15	16,384	8.968	-2.2e-16	False	False
	0	0.5	16	32,768	8.968	2.2e-16	False	False

The `sharpen` exhibit, in its insurer view, shows the same table as a grid: bucket size down the rows, ℓ across the columns, and the score beside the defective flag. A ** marks the starting grid on both axes and surrounds the flag of the selected cell. A missing score or a blank flag marks a cell the line search did not reach.

```
GT(exhibits.exhibit_frames(c, 'sharpen', 'insurer')[0][1])
```

	score		defective	
bs	15	**16	15	**16
1/64	—	4.0041		True
1/32	4.0272	0.0480	True	True
1/16	0.2078	0.2063	True	**False**
1/8	0.8127	0.8127	False	False
**1/4	2.8534	2.8534	False	False
1/2	8.9675	8.9675	False	False

The sharpened program

The string `sharpen_program` is the sharpened program, with its `hints` clause, laid out one clause to a line. Building `c.program` reproduces the sharpened grid.

```
wprint(c.sharpen_program)
d = build(c.program)
print(d.bs, d.log2, d.validation_score)
```

```
agg Cat
  2 claims
  1000 xs 0
  sev lognorm 10 cv 10
  poisson
  hints{log2=16; bs=1/16}
0.0625 16 0.20633928386138262
```

With `execute=False` the probe only diagnoses the grid. The original grid is restored, and the recommendation goes into a `note` rather than a `hints` clause.

```
e = build('agg Cat 2 claims 1000 xs 0 sev lognorm 10 cv 10 poisson')
e.sharpen(execute=False)
wprint(e.sharpen_description)
print(e.bs, e.log2)
wprint(e.program)
```

```
Sharpen: 11 cells in 0.29s, best score 0.206 vs 2.85 at the center, target 0.5.
1 better-scoring cell rejected as defective (mass off the end of the grid).
Recommends bs 1/4 to 1/16, at log2 16; not executed, grid restored.
0.25 16
agg Cat 2 claims 1000 xs 0 sev lognorm 10 cv 10 poisson note{sharpen: probe not
executed, recommends log2 16 and bs 1/16}
```

Sharpening a Portfolio

On a `Portfolio` the call is the same and the score is computed for the total. Here two limited lognormal units on a common grid move from $b = 1/2$ to $1/16$, and the `hints` clause attaches to the portfolio program.

```
p = build('port Cats agg A 2 claims 1000 xs 0 sev lognorm 10 cv 10 poisson '
          'agg B 3 claims 500 xs 0 sev lognorm 20 cv 5 poisson')
p.sharpen()
wprint(p.sharpen_description)
print(p.validation_description)
wprint(p.program)
```

```
Sharpen: 12 cells in 0.26s, best score 0.0544 vs 2.98 at the center, target 0.5.
1 better-scoring cell rejected as defective (mass off the end of the grid).
Moved: bs 1/2 to 1/16, at log2 16.
not unreasonable
port Cats hints{log2=16; bs=1/16} agg A 2 claims 1000 xs 0 sev lognorm 10 cv 10
poisson agg B 3 claims 500 xs 0 sev lognorm 20 cv 5 poisson
```

Refused calls

The line search steps in factors of two, so the argument `bs_limit`, the largest factor by which it may move the bucket, must be a power of two. Any other value raises a `ValueError`.

```
refuse(lambda: c.sharpen(bs_limit=10))
```

```
bs_limit must be a power of two >= 2, got 10
```

Using sliding scale commissions

Sliding scale ceding commissions on an aggregate quota share with the DecL slide clause

Stephen J. Mildenhall

2026-11-01

A **sliding scale commission** is a ceding commission on a proportional treaty, usually a quota share, that falls as the loss ratio of the treaty rises. The reinsurer pays a high commission in a good year and a low one in a bad year, so the cedant keeps part of the variability of the business it cedes. The scale is a set of anchors, a commission rate c_i at a ceded loss ratio λ_i, with the rate decreasing in the loss ratio. With ceded loss A and a fixed ceded premium P_C, the ceded loss ratio is $\lambda = A/P_C$ and the commission is

$$K = c(\lambda)\, P_C,$$

where c interpolates linearly between the anchors and is flat outside them, so the first and last anchors set the maximum and minimum commission.

In DecL the clause `slide <c> at <λ> and ...` decorates one layer of an `aggregate net of` program in place of the flat commission `cede`. The layer needs a `deposit`, `rol`, or `rate` premium to form the loss ratio. The sliding scale is one of the variable features of DecL, with profit commissions, loss corridors, swing rating, and retro rating. A program takes at most one of them. The reinsurance features decorate aggregate covers only, and all take effect in a `pnl` or `xpnl` (see Using pnl and xpnl). The commission is booked in the expense column of the cover, `E` in `economic_ratios_df`, as a random credit to the cedant.

```
aggregate net of 50% po inf xs 0 rate 100% cede 0.25 as QS          # flat
aggregate net of 50% po inf xs 0 rate 100% slide 0.35 at 0.6 and 0.25 at 0.7 and 0.15 at 0.8 as QS
aggregate net of 50% po inf xs 0 rate 100% slide 30% at 55% and 20% at 75% as QS
```

The variable feature pages share one base book: premium 10,000 at a 70% loss ratio, a policy limit of 2,000, lognormal severity with mean 150 and CV 2.5, gamma mixed claim counts with mixing CV 0.25, and expenses of 25% of premium. The setup cell wraps the book in an `xpnl` with a slot for each reinsurance program.

```
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
book = '''agg Book
  10000 premium at 0.7 lr
  2000 xs 0
  sev lognorm 150 cv 2.5 {occurrence}
  mixed gamma 0.25 {aggregate}'''
program = '''xpnl {name}
  inherit premium
  less
```

```
    ''' + book + '''
  less
    0.25 premium expenses'''
def quota_share(name, terms):
    """The base book under a 50% quota share at a 100% rate with the given terms."""
    return build(program.format(name=name, occurrence='',
                 aggregate=f'aggregate net of 50% po inf xs 0 rate 100% {terms} as QS'))
```

```
1.53.0
```

A sliding scale on a 50% quota share

The cover is a 50% quota share of the base book at 100% of the subject premium, so the ceded premium is 5,000 and the ceded loss ratio equals the gross loss ratio, with mean 70%. The scale pays 35% at a ceded loss ratio of 60%, 25% at the expected 70%, and 15% at 80%. The comparison is the same quota share with a flat 25% commission. The parsed terms of the cover are stored in the attribute `variable_terms` of the wrapped aggregate, `engine`.

```
flat = quota_share('Flat', 'cede 0.25')
slide = quota_share('Slide', 'slide 0.35 at 0.6 and 0.25 at 0.7 and 0.15 at 0.8')
print(slide.engine.variable_terms)
qd(slide)
```

```
SlideTerms(anchors=((0.35, 0.6), (0.25, 0.7), (0.15, 0.8)))
PnL('Slide': 2 groups, 6 legs -> 'margin')

                         EX     SD       CV     Skew     P01  Median     P99
Step  Side
Gross Consideration   10000      0        0            10000   10000   10000
      Obligation      -9500   2704 -0.28463 -0.67687  -17071   -9193   -4633
      Margin            500   2704    5.408 -0.67687   -7071     807    5367
QS    Consideration   -5000      0        0            -5000   -5000   -5000
      Obligation     4792.4 990.47  0.20667   1.1004  2816.5    4750  8035.5
      Margin         -207.6 990.47  -4.7711   1.1004 -2183.5    -250  3035.5
      Net             292.4 1753.7   5.9976 -0.47207 -4035.5     557  3183.5
All   Consideration    5000      0        0             5000    5000    5000
      Obligation    -4707.6 1753.7 -0.37253 -0.47207 -9035.5   -4443 -1816.5
      Margin          292.4 1753.7   5.9976 -0.47207 -4035.5     557  3183.5
      Impact         -207.6 990.47  -4.7711   1.1004 -2183.5    -250  3035.5
```

The `QS` step is the cover from the cedant's side: the ceded premium of 5,000 against the recovery and the commission, which together make the obligation row, with mean 4,792. Its `Net` row is the running net result, here the same as `All`. The ledger `economic_df` lists the commission as its own leg, with mean 1,292 and standard deviation 448. In the 1 in 100 bad year, column `κ01`, the commission is at its minimum, 750, and in the 1 in 100 good year, `κ99`, at its maximum, 1,750.

```
GT(slide.economic_df.loc['QS', ['EX', 'SD', 'κ01', 'κ50', 'κ99']], float_format=',.0f')
```

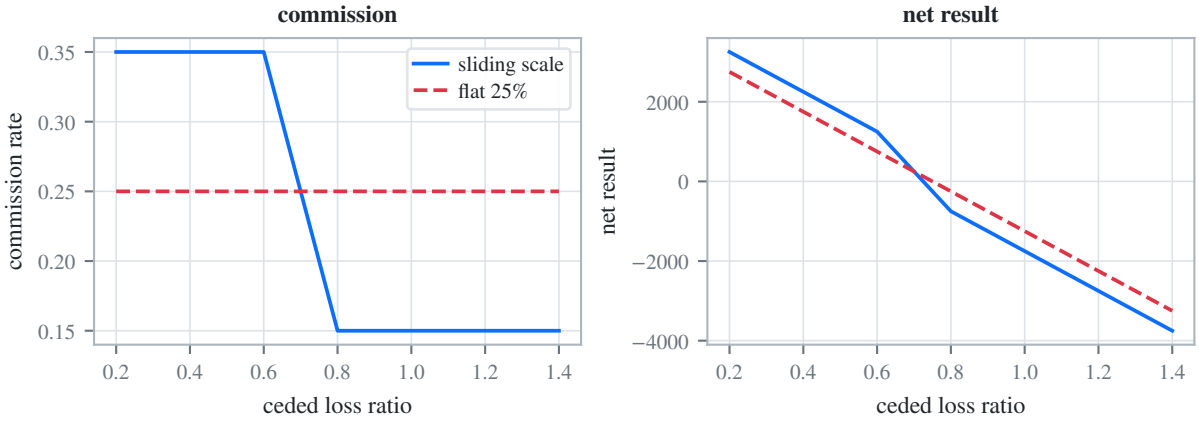
Side	*Label*	EX	SD	κ01	κ50	κ99
Consideration	QS premium	-5,000	0	-5,000	-5,000	-5,000
Obligation	QS recovery	3,500	1,352	7,286	3,346	1,066
	sliding commission	1,292	448	750	1,403	1,750
	Total	4,792	990	8,036	4,750	2,816
Margin	Total	-208	990	3,036	-250	-2,184
	Net	292	1,754	-4,036	557	3,184

The mean commission, 1,292, is 25.85% of the ceded premium, more than the 25% paid at the expected loss ratio. Between the end anchors the rate falls one point per point of loss ratio, so $c(\lambda) = 0.95 - \lambda + (\lambda - 0.8)^+ - (0.6 - \lambda)^+$ and, since $\mathsf{P}\lambda = 0.7$, the mean rate is $\mathsf{P}c(\lambda) = 0.25 + \mathsf{P}(\lambda - 0.8)^+ - \mathsf{P}(0.6 - \lambda)^+$. The loss ratio is skewed to the right, so the floor at 15% cuts off a longer tail than the cap at 35%, and the last two terms add 0.85 points.

Commission and net result against the flat commission

The commission rate and the cedant's net result are both functions of the ceded loss ratio. The gross loss is $10{,}000\lambda$ and the recovery $5{,}000\lambda$, so the net result, premium less expense and ceded premium, plus commission and recovery, less gross loss, is $2{,}500 + 5{,}000(c(\lambda) + \lambda) - 10{,}000\lambda$. The left panel shows the two commission rates and the right panel the two net results.

```
lr = np.linspace(0.2, 1.4, 241)
rates = {'sliding scale': slide.engine.variable_terms.phi(lr), 'flat 25%': np.full_like(lr, 0.25)}
fig, (ax0, ax1) = figure('full', cols=2)
for (k, c), ls in zip(rates.items(), ['-', '--']):
    ax0.plot(lr, c, ls=ls, label=k)
    ax1.plot(lr, 10000 - 2500 - 5000 + 5000 * (c + lr) - 10000 * lr, ls=ls, label=k)
ax0.set(xlabel='ceded loss ratio', ylabel='commission rate', title='commission')
ax1.set(xlabel='ceded loss ratio', ylabel='net result', title='net result')
ax0.legend();
```



Between 60% and 80% the scale takes back one point of commission per point of loss ratio, so the net result falls at twice the slope of the flat cover: the cedant bears the whole movement of the ceded loss ratio in the band, on top of its retained half. Outside the band the two lines are parallel, the sliding scale 500 above the flat cover in good years and 500 below it in bad ones. The table compares the two covers over the whole distribution, with `P01` the 1 in 100 bad result.

```
df = pd.DataFrame({k: [-x.economic_ratios_df.loc['QS', 'E'], x.est_m, x.est_sd, x.q(0.01),
x.cdf(0)]
                   for k, x in {'flat 25%': flat, 'sliding scale': slide}.items()},
                  index=['commission', 'mean', 'SD', 'P01', 'Pr(loss)']).T
df.index.name = 'cover'
GT(df, float_format=',.0f', ratio_cols=['Pr(loss)'])
```

cover	commission	mean	SD	P01	Pr(loss)
flat 25%	1,250	250	1,352	-3,536	38.5%
sliding scale	1,292	292	1,754	-4,036	41.9%

The sliding scale adds 42 to the mean result, from the 0.85 points of extra expected commission, and costs 500 in the 1 in 100 year, where `P01` falls from −3,536 to −4,036. The standard deviation (SD) of the net result rises from 1,352, half the gross 2,704, to 1,754, and the probability of a loss, a result of zero or less, from 38.5% to 41.9%.

Refused programs

A sliding scale replaces the flat commission across its aggregate program, so a `cede` on the same layer or on another aggregate layer is refused. A scale whose commission rises with the loss ratio is not a sliding scale and is refused too.

```
refuse(lambda: quota_share('R1', 'cede 0.25 slide 0.35 at 0.6 and 0.15 at 0.8'),
       lambda: quota_share('R2', 'slide 0.35 at 0.6 and 0.15 at 0.8 and 2000 xs 8000 rate 6% cede
       0.2'),
       lambda: quota_share('R3', 'slide 0.15 at 0.6 and 0.35 at 0.8'))
```

```
DecL: 'slide' replaces the fixed 'cede' commission; give one or the other.
DecL: 'slide' replaces the fixed 'cede' commission across its aggregate program,
so a 'cede' on another layer would not be booked; give one or the other.
SlideTerms: commission must be nonincreasing in loss ratio (a sliding scale),
got ((0.15, 0.6), (0.35, 0.8)).
```

Using `ssev`

Signed severities with ssev, and aggregates or portfolios as severities

Stephen J. Mildenhall

2026-10-30

The DecL keyword `ssev`[1] declares a **signed severity**: it keeps the negative part of the support of a severity, where `sev` moves it onto an atom at zero. So `sev X` is the loss $X^+ = \max(X, 0)$, and `ssev X` has the law of X itself. A negative loss is a gain, so with `ssev` a profit can enter an aggregate. A signed aggregate stays on the loss convention, with `value_type` equal to `loss`: positive values are losses and negative values are gains.

Either keyword also takes an aggregate or a portfolio stored in the recipe base (see Using recipes), as `sev agg.NAME` or `sev port.NAME`. The severity is then the law of the total S of that aggregate or portfolio: its computed probability mass function is used exactly as a `dsev`. With an outer count N the result is $S_1 + \cdots + S_N$, for example a book of policies whose loss per policy is itself a compound.

A layers clause `y xs a` conditions a referenced severity on exceeding the attachment, as it does any severity, so `300 xs 0` turns S into $S \mid S > 0$: the atom $p_0 = \Pr(S = 0)$ is removed and the rest is scaled by $1/(1 - p_0)$. A trailing `!` makes the layer unconditional and keeps the atom; with no layers clause the atom stays as well. For a Poisson count the two are thinnings of each other: λ policies with the atom are $\lambda(1 - p_0)$ policies without it.

```
sev  10 * norm + 5              # a loss: the negative part is clamped to zero
ssev 10 * norm + 5              # signed: the negative part is kept
sev  agg.NAME                   # the total of agg NAME as the severity
300 xs 0 sev agg.NAME !         # limited at 300, the atom at zero kept
ssev port.NAME as "Desk"        # a signed portfolio total, with a label
```

The referenced declaration must fix its own grid with `hints{log2=...; bs=...}`. A reference cannot be modified, so scaling, shifting, picks, and splicing are not available.

```
import warnings
import numpy as np
from scipy import stats as ss
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.0, as are severity references to aggregates and portfolios.

A normal severity under sev and ssev

A normal severity with mean $\mu = 5$ and standard deviation $\sigma = 10$ shows the difference between the two keywords. Under `sev` the mean is $\mathsf{P}X^+ = \sigma\phi(\mu/\sigma) + \mu\Phi(\mu/\sigma) = 6.978$, where ϕ and Φ are the standard normal density and distribution function. Under `ssev` it is 5.

```
a = build('agg Clamped dfreq [1] sev 10 * norm + 5')
b = build('agg Signed dfreq [1] ssev 10 * norm + 5')
qd(a)
qd(b)
print(10 * ss.norm.pdf(0.5) + 5 * ss.norm.cdf(0.5))
```

```
Aggregate object: Clamped. Frequency distribution empirical. Severity
norm, [0, inf), super-exponential right tail. Updated with bucket size
1/256 and log2 = 16. Validation: not unreasonable.

      Mean     SD     CV   Skew P01 Median    P99
X
Freq     1      0      0
Sev  6.978 7.4394 1.0661 1.0139   0      5 28.262
Agg  6.978 7.4394 1.0661 1.0139   0      5 28.262
Aggregate object: Signed. Frequency distribution empirical. Severity
norm, (-inf, inf), super-exponential both tails; signed, support [-inf,
inf]. Updated with bucket size 1/256 and log2 = 16. Validation: not
unreasonable.

      Mean  SD  CV       Skew     P01 Median    P99
X
Freq     1   0   0
Sev      5  10   2 3.4656e-10 -18.262      5 28.262
Agg      5  10   2 3.4656e-10 -18.262      5 28.262
6.97796557401306
```

Both laws put probability $\Phi(-1/2) = 0.309$ at or below zero: the clamped law as an atom at zero, the signed law spread over the negative half line. The signed aggregate still has `value_type` equal to `loss`.

```
print(a.prob_eq_0, a.cdf(0), b.cdf(0), ss.norm.cdf(-0.5), b.value_type)
```

```
0.3086063048423492 0.3086063048423492 0.3086063048416497 0.3085375387259869 loss
```

An aggregate as a severity

An aggregate used as a severity must fix its own grid with `hints`. Our example is the loss on one policy, a Poisson count with mean 1/2 and a lognormal severity limited to 100. Its atom at zero is $p_0 = e^{-1/2}$, the probability of no claim, since a lognormal puts almost no probability near zero.

```
policy = build('agg Policy 0.5 claims 100 xs 0 sev lognorm 50 cv 1 poisson '
               'hints{log2=12; bs=1/4}')
p0 = policy.density_df.p_total.iloc[0]
print(p0, np.exp(-0.5))
```

```
0.6065306597144531 0.6065306597126334
```

A book of 100 policies is built four ways: with no layers clause, with a 300 limit and the default conditional layer, with the same limit and `!`, and with the conditional layer on the thinned count $100(1 - p_0) = 39.35$. The first and third keep the atom, so their severity mean is the policy mean, 21.83. The second removes it, and the mean rises to $21.83/(1 - p_0) = 55.47$. The last two are the same law, with mean 2,182.6 and standard deviation 430.25.

```
books = [build('agg Open 100 claims sev agg.Policy poisson'),
         build('agg Cond 100 claims 300 xs 0 sev agg.Policy poisson'),
         build('agg Uncond 100 claims 300 xs 0 sev agg.Policy ! as "Per policy" poisson'),
         build(f'agg Thin {100 * (1 - p0)} claims 300 xs 0 sev agg.Policy poisson')]
for bk in books:
    print(f'{bk.name:8s} {bk.n:8.3f} {bk.est_sev_m:8.3f} {bk.est_m:10.3f} {bk.est_sd:10.4f}')
```

```
Open      100.000   21.828   2182.808   430.3828
Cond      100.000   55.472   5547.160   685.9122
Uncond    100.000   21.826   2182.637   430.2531
Thin       39.347   55.472   2182.637   430.2531
```

The attribute `pprogram` of an aggregate shows its program as DecL, here with the reference and its label. The label is also available as `labels.severity`.

```
print(books[2].pprogram)
print(books[2].labels.severity)
```

```
agg Uncond
  100 claims
  300 xs 0
  sev agg.Policy ! as "Per policy"
  poisson
Per policy
```

The tail of a referenced severity is the tail of its source, not that of its finite set of atoms. The aggregate `Policy` has a Poisson count, so the sum of two policies is unbounded until a limit applies.

```
print(build('agg Two dfreq [2] sev agg.Policy').bounded,
      build('agg TwoCapped dfreq [2] 300 xs 0 sev agg.Policy !').bounded)
```

```
False True
```

A signed aggregate as a severity

A signed aggregate is used as a severity with `ssev`. Our example is a trading day, a Poisson number of trades with mean 4, each a gain of 10 (entered as a loss of −10) or a loss of 5 or 20. A `dsev` keeps a negative outcome as given, so the aggregate is signed. The method `with_hints()` returns the program with the grid chosen by the build, here bucket size 5, ready to register as a source.

```
day = build('agg Day 4 claims dsev [-10 5 20] [0.3 0.5 0.2] poisson')
wprint(day.with_hints())
day = build(day.with_hints())
```

```
agg Day 4 claims dsev [-10 5 20] [0.3 0.5 0.2] poisson hints{log2=7; bs=5;
normalize=True}
```

A month is the sum of 21 independent days, `ssev agg.Day` with a fixed count. A sum of independent compound Poisson variables with a common severity is compound Poisson, so the month is a compound Poisson with mean count 84. On this lattice, with bucket size 5, the two computations agree to within 2×10^{-14}.

```
month = build('agg Month dfreq [21] ssev agg.Day')
check = build('agg Check 84 claims dsev [-10 5 20] [0.3 0.5 0.2] poisson')
print([float(month.q(p)) for p in [0.01, 0.5, 0.99]],
      [float(check.q(p)) for p in [0.01, 0.5, 0.99]])
print(np.abs(month.density_df.p_total - check.density_df.p_total).max())
```

```
[65.0, 290.0, 540.0] [65.0, 290.0, 540.0]
1.5602102942935403e-14
```

Under plain `sev` the negative outcomes of the source are moved onto zero, with a warning. A day is negative with probability 0.227, and with those outcomes clamped to zero the mean of the month rises from 294 to 357.

```
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter('always')
    clamped = build('agg MonthClamped dfreq [21] sev agg.Day')
wprint(caught[0].message)
print(month.est_m, clamped.est_m)
```

```
agg.Day: a signed source is being used as an unsigned severity; 0.22693 of
probability at negative outcomes is clamped onto the zero atom. Write ssev to
keep the negative support -- a signed source in a loss context usually means the
value conventions do not match.
294.00000000697884 356.54100603545743
```

A portfolio as a severity

A portfolio reference, `sev port.NAME`, uses the distribution of the portfolio total as the severity. On a `port` the `hints` clause goes on the header line, after the name. The mean loss per policy of the book equals the portfolio mean, $0.5 \times 50 + 0.2 \times 200 = 65$.

```
pair = build('port Pair hints{log2=14; bs=1/2} '
             'agg A 0.5 claims sev lognorm 50 cv 1 poisson '
             'agg B 0.2 claims sev lognorm 200 cv 1 poisson')
book = build('agg PairBook 100 claims sev port.Pair poisson')
print(pair.est_m, book.est_m / 100)
```

```
64.99896477679953 64.99891365678513
```

Refused references

A reference to a declaration without a fixed grid is refused, and so is a reference inside a `bivariate`. The first error message includes the `hints` clause to add.

```
build('agg Loose 0.5 claims sev lognorm 50 cv 1 poisson')
refuse('agg Outer 100 claims sev agg.Loose poisson',
       'bivariate BV 25 claims agg A dfreq [1] sev agg.Policy '
       'agg B dfreq [1] sev lognorm 10 cv 1 copula independent poisson')
```

```
Outer: the severity reference 'agg.Loose' needs agg.Loose to pin its own grid,
and it declares no log2 and bs. A reference stands for the distribution that
declaration outputs, so the declaration has to say at what resolution, or the
severity changes with the ambient defaults instead of with the model. Update the
source at the resolution you want and re-register it with its grid pinned,
`build(x.with_hints())`, or add the clause by hand: hints{log2=16; bs=1/4}
BV: a severity reference (sev agg.NAME / sev port.NAME) is not supported in a
'bvagg' body. Declare the compound as its own agg and reference that agg by name
from an ordinary aggregate.
```

Using `stable`

Stable laws with the stable statement and aggregate.stable.Stable

Stephen J. Mildenhall

2026-10-30

An α-stable law is the only possible limit of a normalized sum of independent, identically distributed variables. When the summands have finite variance the limit is the normal, $\alpha = 2$. When they have a power tail $\Pr(X > x) \sim cx^{-\alpha}$ with $0 < \alpha < 2$, as a Pareto severity does, the limit is a stable law with index α. This result is the generalized central limit theorem.

A stable law has four parameters: the index $\alpha \in (0, 2]$, the skewness $\beta \in [-1, 1]$, the scale $\gamma > 0$ and the location δ. Two parameterizations are in common use, and they differ only in the location. In Nolan's S1, the default here and in scipy, the characteristic function is

$$\log \mathsf{P}e^{itX} = -\gamma^\alpha |t|^\alpha \big(1 - i\beta \operatorname{sign}(t) \tan(\pi\alpha/2)\big) + i\delta_1 t$$

for $\alpha \neq 1$, and the locations of a sum add. In S0 the location is $\delta_0 = \delta_1 + \beta\gamma \tan(\pi\alpha/2)$, which is continuous in α. For $1 < \alpha < 2$ the mean is δ_1.

The class `aggregate.stable.Stable`[1] represents a stable law. It can be created with the DecL statement `stable`. Without a name the statement stores nothing. With a name, as in `stable x0 1.5 0`, it stores the law in the recipe base under `stable.x0`, as described in Using `stable` names and arithmetic. It is used to approximate the aggregate loss of a large unit with a Pareto severity by its limit, which has a closed-form mean and tail constant. It is also used to combine the limits of several units with a common tail index. Using `AggregateCHF` with stable laws computes the same laws on a grid, accurately into the far tail.

```
stable 1.5 1 2 0                        # alpha beta scale loc, S1
stable 1.5 1 2 -2 s0                    # the same law in S0
stable 1000 claims sev 100 * lomax 1.5  # the stable limit of a Pareto book
stable x0 1.5 1 2 0                     # the first law, stored as stable.x0
```

```
import numpy as np
from aggregate import build, qd, __version__
from aggregate.stable import Stable
from style import wprint, refuse, figures, figure
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.23.0

The stable statement

Printed, a `Stable` shows the law in both parameterizations, its mean, the probability that it exceeds the mean, and the constant A in its power tail $\Pr(X > x) \sim Ax^{-\alpha}$. The object itself holds the S1 parameters.

```
x = build('stable 1.5 1 2 0')
wprint(x)
x
```

```
stable 1.5 1 2 0 = stable 1.5 1 2 -2 s0, mean 0, Pr(X>mean) 0.333333, Pr(X>x) ~
1.12838 x^-1.5

Stable(alpha=1.5, beta=1, scale=2, loc=0)
```

The S0 statement below describes the same law as the S1 statement above. The scale and location are optional, with defaults 1 and 0.

```
wprint(build('stable 1.5 1 2 -2 s0'))
wprint(build('stable 1.5 1'))
```

```
stable 1.5 1 2 0 = stable 1.5 1 2 -2 s0, mean 0, Pr(X>mean) 0.333333, Pr(X>x) ~
1.12838 x^-1.5
stable 1.5 1 1 0 = stable 1.5 1 1 -1 s0, mean 0, Pr(X>mean) 0.333333, Pr(X>x) ~
0.398942 x^-1.5
```

When $\beta = 1$ and $1 < \alpha < 2$ the left tail is light and the mean is the $1/\alpha$ quantile, so $\Pr(X > \mathrm{P}X) = 1 - 1/\alpha$, here 1/3. A law with $\beta < 1$ has a power tail on both sides, and for $\alpha \le 1$ the mean does not exist.

```
wprint(build('stable 1.2 0'))
wprint(build('stable 0.5 1'))
```

```
stable 1.2 0 1 0 = stable 1.2 0 1 0 s0, mean 0, Pr(X>mean) 0.5, Pr(X>x) ~
0.277958 x^-1.2, Pr(X<-x) ~ 0.277958 x^-1.2
stable 0.5 1 1 0 = stable 0.5 1 1 1 s0, mean infinite, Pr(X>loc) 1, Pr(X>x) ~
0.797885 x^-0.5
```

Sums and scaling of stable laws

Independent stable laws with a common index α add to a stable law with the same index: γ^α adds, β is the γ^α-weighted average, and in S1 the locations add. Scaling, negation and shifting stay in the family. For a real $a \neq 0$, aX has skewness $\mathrm{sign}(a)\beta$, scale $|a|\gamma$ and location $a\delta_1$, less $(2/\pi)\beta\gamma a \log|a|$ when $\alpha = 1$. The class `Stable` implements these rules with the operators `+`, `-`, `*` and `/`, and Using `stable` names and arithmetic shows the same operations in DecL. Each operand is an independent copy, so `sum([x1, x1, x1])` is the sum of three independent copies of `x1` and has a different scale from `3 * x1`. The form `3 @ x1` returns the same sum.

```
x1 = Stable(1.5, 1, 2, 10)
x2 = Stable(1.5, -0.5, 1, -3)
wprint(x1 + x2)
wprint(3 * x1 - 5)
wprint(sum([x1, x1, x1]))
wprint(3 @ x1)
```

```
stable 1.5 0.608194 2.44726 7 = stable 1.5 0.608194 2.44726 5.51159 s0, mean 7,
Pr(X>mean) 0.384045, Pr(X>x) ~ 1.22811 x^-1.5, Pr(X<-x) ~ 0.299207 x^-1.5
stable 1.5 1 6 25 = stable 1.5 1 6 19 s0, mean 25, Pr(X>mean) 0.333333, Pr(X>x)
~ 5.86323 x^-1.5
stable 1.5 1 4.16017 30 = stable 1.5 1 4.16017 25.8398 s0, mean 30, Pr(X>mean)
0.333333, Pr(X>x) ~ 3.38514 x^-1.5
stable 1.5 1 4.16017 30 = stable 1.5 1 4.16017 25.8398 s0, mean 30, Pr(X>mean)
0.333333, Pr(X>x) ~ 3.38514 x^-1.5
```

The characteristic function of a sum of independent variables is the product of their characteristic functions. On a grid of t values the two sides differ only by rounding.

```
t = np.linspace(-2, 2, 9)
print(np.abs((x1 + x2).chf(t) - x1.chf(t) * x2.chf(t)).max())
```

```
1.1857187100668868e-16
```

The stable limit of a Pareto book

Take a Poisson count with mean λ and a severity with $\Pr(X > x) = \theta^\alpha x^{-\alpha}(1 + o(1))$, $1 < \alpha < 2$. For large λ the aggregate is approximately $S(\alpha, 1, \gamma, \lambda \mathrm{P}X; 1)$ with $\gamma^\alpha = \lambda\theta^\alpha\Gamma(1-\alpha)\cos(\pi\alpha/2)$, which has exactly the aggregate's mean and right tail constant $\lambda\theta^\alpha$. The `stable` statement computes it from the count and severity.

```
s = build('stable 1000 claims sev 100 * lomax 1.5')
wprint(s)
```

```
stable 1.5 1 18452.7 200000 = stable 1.5 1 18452.7 181547 s0, mean 200000,
Pr(X>mean) 0.333333, Pr(X>x) ~ 1e+06 x^-1.5
```

The aggregate itself can be computed for comparison. The severity has infinite variance, so the grid is given explicitly. Validation fails the means because the grid truncates the tail: the computed aggregate mean is about 198,000 against 200,000.

```
a = build('agg Pareto 1000 claims sev 100 * lomax 1.5 poisson', bs=10, log2=18)
qd(a)
```

```
Aggregate object: Pareto. Frequency distribution poisson. Severity
lomax, [0, inf), power-law right tail. Updated with bucket size 10 and
log2 = 18. Validation: fails sev mean, agg mean.

           Mean     SD       CV     Skew        P01     Median        P99
X
Freq       1000 31.623 0.031623 0.031623
Sev      198.08 2176.8   10.989   411.09          0         60       2050
Agg  1.9808e+05  69002  0.34835   12.724 1.4284e+05 1.8618e+05 4.1113e+05
```

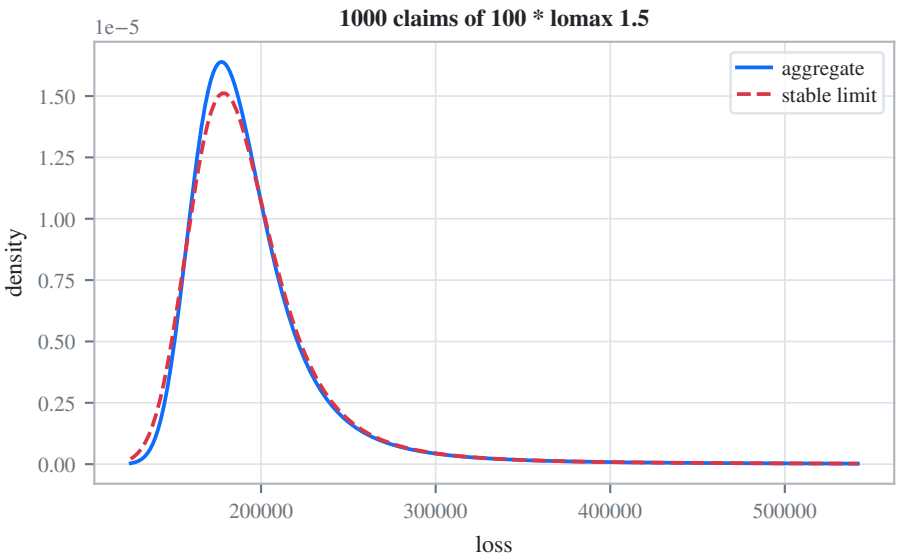
The quantiles of the aggregate and of its stable limit agree to within 4% at the 1% level and to within 1.5% from the 10% level up. The stable limit is lower at the bottom of the distribution, where the aggregate is still some way from its limit.

```
ps = [0.01, 0.1, 0.5, 2/3, 0.9, 0.99]
for p in ps:
    print(f'{p:6.3f} {a.q(p):12,.0f} {s.ppf(p):12,.0f}')
```

```
0.010      142,840      137,793
0.100      159,120      156,982
0.500      186,180      186,775
0.667      198,800      199,879
0.900      237,970      239,595
0.990      411,130      415,050
```

The densities are close across the body of the distribution. The stable limit puts a little more probability in the left tail, as the 1% quantiles show.

```
xs = np.linspace(s.ppf(0.001), s.ppf(0.995), 200)
fig, ax = figure('three-quarter')
df = a.density_df.loc[xs[0]:xs[-1]]
ax.plot(df.index, df.p_total / a.bs, label='aggregate')
ax.plot(xs, s.pdf(xs), ls='--', label='stable limit')
ax.set(xlabel='loss', ylabel='density', title='1000 claims of 100 * lomax 1.5')
ax.legend();
```



Distribution values and the far tail

The methods `pdf`, `cdf`, `sf`, `ppf`, `isf` and `rvs` of `Stable` call `scipy.stats.levy_stable`, which is accurate in the body of the distribution but not in the far tail. There the closed-form tail $Az^{-\alpha}$ is the better guide. At $z = 5430$ the scipy survival function returns zero, and `ppf(1 - 1e-6)` returns 659, a point with survival probability 2.4×10^{-5}, where the tail approximation gives about 5,420.

```
u = Stable(1.5, 1)
for z in [100, 659, 5430]:
    print(f'{z:6d} {u.sf(z):12.4e} {u.tail_constant * z ** -1.5:12.4e}')
print(u.ppf(1 - 1e-6), (u.tail_constant / 1e-6) ** (1 / 1.5))
```

```
   100   3.9894e-04   3.9894e-04
   659   2.3582e-05   2.3582e-05
  5430   0.0000e+00   9.9703e-07
659.2325672109375 5419.260701392888
```

Refused programs

An index outside $(0, 2]$ is refused, as is a parameterization other than `s0` or `s1`. The limit form needs a Pareto or Lomax severity with tail index strictly between 1 and 2, so a finite-variance Lomax and a lognormal are refused too.

```
refuse('stable 2.5 0',
       'stable 1.5 1 2 0 s2',
       'stable 1000 claims sev 100 * lomax 2.5',
       'stable 1000 claims sev lognorm 10 cv 1')
```

```
Stable: alpha must be in (0, 2], got 2.5
stable: the statement is 'stable alpha beta [scale [loc]] [s0|s1]'; the last
word must be s0 or s1, got 's2'
stable: the limit theorem form needs a plain pareto or lomax severity, `stable
<count> claims sev <scale> * lomax <alpha>`, but found the tail index 2.5, which
must be strictly between 1 and 2.
stable: the limit theorem form needs a plain pareto or lomax severity, `stable
<count> claims sev <scale> * lomax <alpha>`, but found severity 'lognorm'.
```

Using `stable` names and arithmetic

Named stable and tweedie laws, and arithmetic on stable references

Stephen J. Mildenhall

2026-10-30

A sum of independent stable laws with a common index α is a stable law, and so is a scaled or shifted stable law. Using `stable` describes the four parameters and the two parameterizations, and states the rules for the parameters of a sum, a multiple, and a shift. The class `aggregate.stable.Stable` computes these combinations in Python, and DecL reaches them through names. A `stable` statement with a name[1] stores the law in the recipe base, and the reference `stable.NAME` returns it. References take arithmetic[2], which is used to combine the stable limits of several units. A `tweedie` statement takes the same name and reference but no arithmetic, because a sum of independent Tweedie laws is a Tweedie law only when they have the same gamma severity.

```
stable x0 1.5 0                          # a name, then any form of the statement
stable x1 1.5 0.85
stable Book as "Pareto book" 1000 claims sev 100 * lomax 1.5
tweedie t0 1.5 100 2

3 * stable.x0                            # an anonymous law, not stored
stable.x0 + 2 * stable.x1 + 3
stable z stable.x0 + 2 * stable.x1 + 3   # the same combination, stored as stable.z
3 @ stable.x0                            # the sum of three independent copies
```

```
from aggregate import build, __version__
from style import wprint, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

Named stable and tweedie laws

A named law is a `stable` or `tweedie` statement with a name after the keyword, followed by an optional `as` label and any form of the statement, with optional `note`, `tags`, and `hints`. Printed, it starts with the name, and each form after the colon is valid DecL.

```
wprint(build('stable x0 1.5 0'))
wprint(build('stable x1 1.5 0.85'))
wprint(build('stable x2 1.5 -0.5'))
wprint(build('stable Book as "Pareto book" 1000 claims sev 100 * lomax 1.5'))
wprint(build('tweedie t0 1.5 100 2'))
```

[1] From aggregate 1.42.0
[2] From aggregate 1.43.0

```
x0: stable 1.5 0 1 0 = stable 1.5 0 1 0 s0, mean 0, Pr(X>mean) 0.5, Pr(X>x) ~
0.199471 x^-1.5, Pr(X<-x) ~ 0.199471 x^-1.5
x1: stable 1.5 0.85 1 0 = stable 1.5 0.85 1 -0.85 s0, mean 0, Pr(X>mean)
0.350502, Pr(X>x) ~ 0.369022 x^-1.5, Pr(X<-x) ~ 0.0299207 x^-1.5
x2: stable 1.5 -0.5 1 0 = stable 1.5 -0.5 1 0.5 s0, mean 0, Pr(X>mean) 0.598389,
Pr(X>x) ~ 0.0997356 x^-1.5, Pr(X<-x) ~ 0.299207 x^-1.5
Pareto book (Book): stable 1.5 1 18452.7 200000 = stable 1.5 1 18452.7 181547
s0, mean 200000, Pr(X>mean) 0.333333, Pr(X>x) ~ 1e+06 x^-1.5
t0: tweedie 1.5 100 2 = tweedie 1.5 100 cv 0.447214 = tweedie 10 claims sev
gamma 10 cv 1, Pr(X=0) 4.53999e-05
```

The recipe base stores numbers whichever form was written: the four S1 parameters for `stable`, and *p*, the mean, and the dispersion for `tweedie`. The program is stored with the entry, so the limit form of `Book` remains readable while its stored law is four numbers.

```
book = build.recipe('Book')
wprint(book.program)
print({k: float(book.spec[k]) for k in ('alpha', 'beta', 'scale', 'loc')})
```

```
stable Book as "Pareto book" 1000 claims sev 100 * lomax 1.5
{'alpha': 1.5, 'beta': 1.0, 'scale': 18452.701486440274, 'loc': 200000.0}
```

A later statement refers to a named law as `stable.x0` or `tweedie.t0`, the same dotted form as `agg.X` and `sev.X`. Either reference alone returns the stored law, with its name and label. The bare name does too when it is unique across kinds.

```
print(repr(build('stable.x0')))
print(repr(build('x0')))
print(build('tweedie.t0').dispersion)
```

```
Stable(alpha=1.5, beta=0, scale=1, loc=0, name='x0')
Stable(alpha=1.5, beta=0, scale=1, loc=0, name='x0')
2.0
```

Arithmetic on stable references

Stable references take `+` and `-` between laws, `+`, `-`, `*`, and `/` with a number, unary minus, and `@`. An expression on its own returns an anonymous `Stable` and stores nothing. On the numeric side anything a DecL expression accepts works, and a `+`, `-`, or `*` between numbers needs parentheses, as it does everywhere in DecL.

```
print(repr(build('3 * stable.x0')))
print(repr(build('stable.x1 + stable.x2')))
print(repr(build('stable.x0 + 2 * stable.x1 + 3')))
print(repr(build('(1 + 2) * stable.x0 / 6 - 1')))
```

```
Stable(alpha=1.5, beta=0, scale=3, loc=0)
Stable(alpha=1.5, beta=0.175, scale=1.5874010519681994, loc=0)
Stable(alpha=1.5, beta=0.6279767062808198, scale=2.4472608147714756, loc=3)
Stable(alpha=1.5, beta=0, scale=0.5, loc=-1)
```

A stored combination is written `stable z <expression>`. A reference resolves when the statement is parsed, so `z` holds four numbers and redefining `x0` afterward leaves `z` unchanged.

```
z = build('stable z stable.x0 + 2 * stable.x1 + 3')
wprint(z)
build('stable x0 1.5 0 9')
print(build('stable.z').scale == z.scale)
build('stable x0 1.5 0');
```

```
z: stable 1.5 0.627977 2.44726 3 = stable 1.5 0.627977 2.44726 1.46318 s0, mean
3, Pr(X>mean) 0.381008, Pr(X>x) ~ 1.24322 x^-1.5, Pr(X<-x) ~ 0.2841 x^-1.5
True
```

Independent copies of a stable law

The arithmetic operations act on laws, not on random variables, and each occurrence of a reference is an independent copy. As a result `stable.x0 + stable.x0` is different from `2 * stable.x0`. At $\alpha = 3/2$ the first has scale $2^{2/3} = 1.5874$ and the second has scale 2, because the scale of a sum of independent copies grows more slowly than the number of copies. Likewise `stable.x0 - stable.x0` is a symmetric stable law, not zero.

```
print(repr(build('stable.x0 + stable.x0')))
print(repr(build('2 * stable.x0')))
print(repr(build('stable.x0 - stable.x0')))
```

```
Stable(alpha=1.5, beta=0, scale=1.5874010519681994, loc=0)
Stable(alpha=1.5, beta=0, scale=2, loc=0)
Stable(alpha=1.5, beta=0, scale=1.5874010519681994, loc=0)
```

The form `t @ stable.x` is the sum of t independent copies, with scale $t^{1/\alpha}\gamma$ and location $t\delta$. Its characteristic function is φ^t, so t need not be an integer, and the result is the law at time t of the Lévy process with `stable.x` at time 1. The operator `@` is the same as in `2 @ agg.X`, which doubles a claim count. The last line below adds two laws at time 1/2, which returns `stable.x1`.

```
print(repr(build('2 @ stable.x1')))
print(repr(build('stable.x1 + stable.x1')))
print(repr(build('0.5 @ stable.x1 + 0.5 @ stable.x1')))
```

```
Stable(alpha=1.5, beta=0.85, scale=1.5874010519681994, loc=0)
Stable(alpha=1.5, beta=0.85, scale=1.5874010519681994, loc=0)
Stable(alpha=1.5, beta=0.85, scale=1, loc=0)
```

Limit laws of two units

Two units with Pareto severities and a common tail index combine through their stable limits. The sum has the sum of the means. Its right tail constant is the sum of the two values of $\lambda\theta^\alpha$, which gives a closed-form check. The difference has both tails, with the larger on the side of the unit with the larger $\lambda\theta^\alpha$.

```
build('stable A 1000 claims sev 100 * lomax 1.5')
build('stable B 400 claims sev 250 * lomax 1.5')
total = build('stable.A + stable.B')
wprint(total)
print(f'{total.tail_constant:,.2f} {1000 * 100 ** 1.5 + 400 * 250 ** 1.5:,.2f}')
wprint(build('stable.A - stable.B'))
```

```
stable 1.5 1 34721.7 400000 = stable 1.5 1 34721.7 365278 s0, mean 400000,
Pr(X>mean) 0.333333, Pr(X>x) ~ 2.58114e+06 x^-1.5
2,581,138.83 2,581,138.83
stable 1.5 -0.225148 34721.7 0 = stable 1.5 -0.225148 34721.7 7817.53 s0, mean
0, Pr(X>mean) 0.546994, Pr(X>x) ~ 1e+06 x^-1.5, Pr(X<-x) ~ 1.58114e+06 x^-1.5
```

Stable laws in the recipe library

The recipe library included with aggregate (see Using recipes) holds four stable laws under their classical names: `stable.Cauchy` ($\alpha = 1, \beta = 0$), `stable.Levy` ($\alpha = 1/2, \beta = 1$), `stable.Landau` ($\alpha = 1, \beta = 1$), and `stable.Holtsmark` ($\alpha = 3/2, \beta = 0$). They are available in any session and to a client that resolves library entries only, such as a spreadsheet, which has no access to a law declared in another cell.

```
for name in ('Cauchy', 'Levy', 'Landau', 'Holtsmark'):
    wprint(build(f'stable.{name}'))
print(repr(build('stable.Holtsmark + 2 * stable.Holtsmark')))
```

```
Cauchy: stable 1 0 1 0 = stable 1 0 1 0 s0, mean undefined, Pr(X>loc) 0.5,
Pr(X>x) ~ 0.31831 x^-1, Pr(X<-x) ~ 0.31831 x^-1
Levy: stable 0.5 1 1 0 = stable 0.5 1 1 1 s0, mean infinite, Pr(X>loc) 1,
Pr(X>x) ~ 0.797885 x^-0.5
Landau: stable 1 1 1 0 = stable 1 1 1 0 s0, mean infinite, Pr(X>x) ~ 0.63662
x^-1
Holtsmark: stable 1.5 0 1 0 = stable 1.5 0 1 0 s0, mean 0, Pr(X>mean) 0.5,
Pr(X>x) ~ 0.199471 x^-1.5, Pr(X<-x) ~ 0.199471 x^-1.5
Stable(alpha=1.5, beta=0, scale=2.4472608147714756, loc=0)
```

Refused programs

A law must be named to enter arithmetic, because `stable 1.5 -0.5` would otherwise have two readings, and DecL has no variables, so `3 * stable 1.5 0` and `x0 = stable 1.5 0` are parse errors. A name may contain a minus sign, so a subtraction needs a space before the minus, and the message for a missing name of that shape shows the form with spaces. The laws in a sum must have a common α. Scaling by zero, division by zero, and zero copies are refused. A `tweedie` reference takes no arithmetic, and a `chfagg` takes the parameters of a stable law but not a reference to one. A name held under two kinds must be written with its prefix.

```
build('agg x0 dfreq [1] dsev [1]')
refuse('3 * stable 1.5 0',
       'x0 = stable 1.5 0',
       'stable.x1-stable.x2',
       'stable.x0 + stable.Cauchy',
       '0 * stable.x1',
       'stable.x1 / 0',
       '0 @ stable.x1',
       '2 * tweedie.t0',
       'chfagg C stable.x1',
       'x0')
```

```
DecL parse error at line 1, column 5: Unexpected 'stable'.
DecL parse error at line 1, column 1: Unexpected 'x0'.
no recipe named 'x1-stable.x2' of kind 'stable'. A name may contain a minus
sign, so stable.x1-stable.x2 is one name. To subtract, write spaces: stable.x1 -
stable.x2
```

```
Stable: only laws with a common alpha add to a stable law, got 1.5 and 1.0
Stable: scaling by zero gives a point mass, not a stable law
Stable: division by zero does not give a stable law
Stable: the number of independent copies in t @ X must be positive and finite,
got 0.0
DecL parse error at line 1, column 5: Unexpected 'tweedie.t0'.
DecL parse error at line 1, column 10: Unexpected 'stable.x1'. Did you mean:
stable?
'x0' is ambiguous across kinds (agg, stable); write agg.x0 or stable.x0, or pass
kind= to choose. Shipped library names are unique across kinds.
```

Using swing rating

Swing-rated aggregate reinsurance, with a ceded premium set by the ceded loss between a minimum and a maximum

Stephen J. Mildenhall

2026-11-01

A **swing-rated** cover sets its ceded premium after the treaty year from the reinsurer's own losses. With ceded loss A, a basic premium b, a loss conversion multiplier m, and a minimum ℓ and maximum u, the ceded premium is

$$p(A) = \min\big(\max(b + mA,\ \ell),\ u\big).$$

Swing plans are used on working layers and aggregate covers whose losses are predictable enough that the cedant is content to pay them back. Between the minimum and the maximum, each unit of ceded loss raises the premium by m, so the cover gives real protection only in the years when the premium reaches its maximum. The minimum defaults to the basic and the maximum to no cap.

The clause `swing basic <b> lcm <m> [min <lo>] [max <hi>]` takes the place of the `deposit`, `rol`, or `rate` premium clause on an aggregate layer. A `pnl` or `xpnl` books the result (see Using pnl and xpnl). A program has at most one variable feature: swing, a sliding scale, a profit commission, or a loss corridor. Swing on an occurrence layer is refused; there, reinstatements give the random ceded premium.

```
aggregate net of 4000 xs 8000 swing basic 200 lcm 1.1 min 500 max 2500
4000 xs 8000 swing basic 200 lcm 1.1 max 2500     # minimum 200, the basic
4000 xs 8000 swing basic 0 lcm 1.25 min 500        # no maximum
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.60.1
```

A swing-rated stop loss on the base book

The variable feature pages share one base book: premium 10,000 at a 70% loss ratio, a policy limit of 2,000, lognormal severity with mean 150 and CV 2.5, gamma mixed claim counts with mixing CV 0.25, and expenses of 25% of premium. The cover here is a stop loss of 4,000 xs 8,000, from an 80% to a 120% loss ratio, with swing premium $200 + 1.1A$, at least 500 and at most 2,500.

```
book = '''agg Book
  10000 premium at 0.7 lr
  2000 xs 0
  sev lognorm 150 cv 2.5 {occurrence}
  mixed gamma 0.25 {aggregate}'''
program = '''xpnl {name}
  inherit premium
  less
    ''' + book + '''
  less
    0.25 premium expenses'''
layer = 'aggregate net of 4000 xs 8000 {} as "Stop loss"'
s = build(program.format(name='Swing', occurrence='', aggregate=layer.format(
    'swing basic 200 lcm 1.1 min 500 max 2500')))
qd(s)
gross = s.engine
print(f'premium at the minimum {gross.cdf(8000 + 300 / 1.1):.4f}, '
      f'at the maximum {gross.sf(8000 + 2300 / 1.1):.4f}')
```

```
PnL('Swing': 2 groups, 5 legs -> 'margin')

                            EX     SD       CV     Skew    P01  Median   P99
Step      Side
Gross     Consideration   10000      0        0           10000   10000 10000
          Obligation      -9500   2704 -0.28463 -0.67687 -17071   -9193 -4633
          Margin            500   2704    5.408 -0.67687  -7071     807  5367
Stop loss Consideration -902.87  733.2 -0.81208  -1.4843  -2500    -500  -500
          Obligation      607.6 1160.2   1.9094   1.8962      0       0  4000
          Margin        -295.26 497.71  -1.6856   2.8642   -500    -500  1500
          Net            204.73 2352.5    11.49 -0.27905  -5571     307  4867
All       Consideration  9097.1  733.2 0.080597  -1.4843   7500    9500  9500
          Obligation    -8892.4 1841.3 -0.20706  0.20371 -13071   -9193 -4633
          Margin         204.73 2352.5    11.49 -0.27905  -5571     307  4867
          Impact        -295.26 497.71  -1.6856   2.8642   -500    -500  1500
premium at the minimum 0.7113, at the maximum 0.1297
```

The `Stop loss` step shows a ceded loss with mean 607.60, which is zero in more than half of all years, and a ceded premium with mean 902.87 and standard deviation (SD) 733.20. The premium is at the minimum when the ceded loss is at most $300/1.1 = 272.7$, in 71.1% of years, and at the maximum when the ceded loss exceeds $2{,}300/1.1 = 2{,}090.9$, in 13.0% of years. Without the collar the mean premium would be $200 + 1.1 \times 607.60 = 868.36$; the minimum adds more than the maximum removes.

Swing rating against a fixed premium

A fixed premium of 902.87, the mean swing premium, on the 4,000 xs 8,000 stop loss gives the same mean results as the swing plan, so the comparison isolates the effect of the swing on the spread. The table shows the mean and SD of the ceded premium, the SD of the cover's result (premium less recovery), and the mean, SD, and 1 in 100 result `P01` of the cedant's net result, with the uncovered book for reference.

```
premium = -s.summary_df.loc[('Stop loss', 'Consideration'), 'EX']
f = build(program.format(name='Fixed', occurrence='',
                         aggregate=layer.format(f'deposit {premium:.2f}')))
rows = {}
for k, x in {'fixed': f, 'swing': s}.items():
    t = x.summary_df
    rows[k] = [-t.loc[('Stop loss', 'Consideration'), 'EX'],
```

```
              t.loc[('Stop loss', 'Consideration'), 'SD'],
              t.loc[('Stop loss', 'Margin'), 'SD'],
              *t.loc[('All', 'Margin'), ['EX', 'SD', 'P01']]]
g = s.summary_df.loc[('Gross', 'Margin'), ['EX', 'SD', 'P01']]
rows = {'no cover': [0, 0, 0, *g], **rows}
df = pd.DataFrame.from_dict(rows, orient='index', columns=[
    'premium', 'premium SD', 'cover SD', 'net', 'net SD', 'net P01'])
df.index.name = 'ceded premium'
GT(df, float_format=',.2f')
```

ceded premium	premium	premium SD	cover SD	net	net SD	net P01
no cover	0.00	0.00	0.00	500.00	2,703.99	-7,071.00
fixed	902.87	0.00	1,160.18	204.73	1,841.30	-3,973.87
swing	902.87	733.20	497.71	204.73	2,352.46	-5,571.00

Swing rating returns most of the volatility to the cedant. Inside the collar each unit of ceded loss raises the premium by 1.1, so the cedant pays back more than it recovers, and the cover gives real protection only above a ceded loss of 2,090.9, where the premium stops at its maximum. The SD of the cover's result falls from 1,160.18 under the fixed premium to 497.71 under the swing, and the cedant's net SD rises from 1,841.30 to 2,352.46, against 2,703.99 with no cover. The 1 in 100 net result is −5,571 under the swing, against −3,974 under the fixed premium and −7,071 with no cover.

Refused programs

Swing is refused on an occurrence layer, alongside a `deposit`, `rol`, or `rate` clause, alongside `cede` on its own layer, and on a second layer, since a program has at most one variable feature.

```
swing = 'swing basic 200 lcm 1.1 min 500 max 2500'
refuse(program.format(name='S1', aggregate='',
                      occurrence=f'occurrence net of 1000 xs 1000 {swing}'),
       program.format(name='S2', occurrence='',
                      aggregate=f'aggregate net of 4000 xs 8000 rol 0.2 {swing}'),
       program.format(name='S3', occurrence='',
                      aggregate=f'aggregate net of 4000 xs 8000 cede 0.1 {swing}'),
       program.format(name='S4', occurrence='',
                      aggregate=f'aggregate net of 4000 xs 8000 {swing} and '
                                f'4000 xs 12000 {swing}'))
```

```
DecL: variable-rating features (swing / slide / pc / corridor) decorate
aggregate reinsurance only. On an occurrence layer, reinstatements give a random
ceded premium.
DecL: 'swing' supplies the ceded premium and replaces the deposit / rol / rate
clause; give one or the other.
DecL: 'swing' supplies the ceded premium and the commission; give it without
'cede'.
DecL: at most one variable-rating feature per program; found 2.
```

Using tail classes

Tail thickness classes and the FFT grid they size

Stephen J. Mildenhall

2026-10-30

An aggregate is computed on $n = 2^\ell$ buckets of size b, with $b =$ `bs` and $\ell =$ `log2`, which for a non-negative law cover the window $[0, nb)$. For a given ℓ a smaller bucket gives finer resolution and a shorter window (Using FFT windows). How far the window must reach depends on how fast the right tail decays, and `aggregate` assigns every tail a **tail class** before it sizes the grid.

Write $H(x) = -\log \Pr(X > x)$ for the cumulative hazard. An unbounded tail is super-exponential when $H(x)/x \to \infty$, exponential when $H(x)/x \to \lambda \in (0, \infty)$, subexponential when $H(x)/x \to 0$ and $H(x)/\log x \to \infty$, and power-law when $H(x)/\log x \to \alpha$. Together with bounded, these are the five tail classes, ordered from lightest to heaviest. The aggregate class is the heavier of the frequency and severity classes. For a subexponential or heavier severity the aggregate tail follows the single big jump, $\Pr(S > x) \sim \mathsf{P}N \Pr(X > x)$: one large claim on an otherwise typical total.

The tail class is determined by the frequency and severity families and any bound a limit imposes, never by a computed density, so it is available before `update()`. The sizer starts from the window of a three-moment fit holding $1 - 10^{-12}$ of the aggregate. For a thick right tail (subexponential or power-law) it also computes the single big jump reach $\mathsf{P}S - \mathsf{P}X + q_X(p^{**})$, where q_X is the severity quantile function and $p^{**} = 1 - 10^{-12}/\mathsf{P}N$. The sizer extends the window to that reach when it fits in 2^ℓ buckets of the same size, and otherwise keeps the window and records the probability it clips. For a thin tail the reach is not used.

Tail classes[1] are computed for an `Aggregate` and for a `Portfolio`, with no DecL of their own. The examples are three aggregates of 20 Poisson claims whose gamma, lognormal and Lomax severities share mean 100 and CV 1.5.

```
agg Gamma 20 claims sev gamma 100 cv 1.5 poisson
agg Lognorm 20 claims sev lognorm 100 cv 1.5 poisson
agg Lomax 20 claims sev 260 * lomax 3.6 poisson
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, figures
figures()
print(__version__)
```

```
1.43.1
```

[1] From aggregate 1.0

Tail classes before update

The attribute `tail_class` is the triple of frequency, severity and aggregate tail classes. The aggregates are built with `update=False`, so there is no grid yet, and `bs_window_df` and the return period table `tail_df` are `None`.

```
programs = {'gamma': 'agg Gamma 20 claims sev gamma 100 cv 1.5 poisson',
            'lognorm': 'agg Lognorm 20 claims sev lognorm 100 cv 1.5 poisson',
            'lomax': 'agg Lomax 20 claims sev 260 * lomax 3.6 poisson'}
books = {k: build(v, update=False) for k, v in programs.items()}
print(books['gamma'].bs_window_df, books['gamma'].tail_df)
GT(pd.DataFrame({k: [str(c) for c in a.tail_class] for k, a in books.items()},
                index=pd.Index(['freq', 'sev', 'agg'], name='component')))
```

```
None None
```

component	gamma	lognorm	lomax
freq	super-exponential	super-exponential	super-exponential
sev	exponential	subexponential	power-law
agg	exponential	subexponential	power-law

The table `tail_behavior_df` has a row for the frequency, one for each severity component, and one for the aggregate. Its columns are the support (`min`, `max`), the class of each tail (`left_tail`, `right_tail`), and for the aggregate a `concentrated` flag (CV under 0.1) and the CV; a few are shown.

```
GT(books['lomax'].tail_behavior_df[['family', 'right_tail', 'cv', 'note']])
```

component	family	right_tail	cv	note
frequency	poisson	super-exponential	—	claim count
comp0	lomax	power-law	—	power-law, alpha=3.6
aggregate	severity-driven	power-law	0.4031	power-law, alpha=3.6

The string `tail_description` gives the same information in three aligned lines, and `tail_explanation` gives it in prose, with the mechanism behind the aggregate tail. For the Lomax the mechanism is the single big jump, and the aggregate has the same power-law index as the severity, $\alpha = 3.6$.

```
wprint(books['lomax'].tail_description)
wprint(books['lomax'].tail_explanation)
```

```
frequency tail          poisson, count [0, inf), super-exponential right tail
severity tail           lomax, [0, inf), power-law right tail
aggregate tail          [0, inf), power-law right tail; not concentrated
                        (cv=0.403) [power-law, alpha=3.6]
Poisson frequency on counts [0, inf) (super-exponential right tail). Lomax
severity [0, inf), power-law right tail. The aggregate is [0, inf), power-law
right tail. Its right tail follows the severity by the single big jump P(S>x) ~
E[N]*P(X>x). Power-law, alpha=3.6. It is not concentrated: cv ~ 0.403.
```

Windows for three tail classes

The table `bs_window_df` has a row for each sizing method that applied and a `used` row for the realized window (Using FFT windows). The `selected` column marks the method whose bucket was taken, `W` is the width of each window, and `coverage` the probability it is sized to hold, adjusted by p^{**} for `sbj` (shown as `E[N]-adj`). The `clipped` column estimates the probability beyond the top of the window.

```
for a in books.values():
    a.update()
cols = ['selected', 'W', 'bs', 'log2', 'coverage', 'clipped']
GT(pd.concat({k: a.bs_window_df[cols] for k, a in books.items()},
             names=['severity', 'method']), float_format=',.6g')
```

severity	*method*	selected	W	bs	log2	coverage	clipped
gamma	moment	True	15,038.9	0.25	16	1-1e-12	—
	sbj	False	30,971.4	0.5	16	E[N]-adj 1-1e-12	—
	used	False	16,384	0.25	16	1-1e-12	—
lognorm	moment	True	29,492.3	0.5	16	1-1e-12	—
	sbj	False	180,738	4	16	E[N]-adj 1-1e-12	—
	used	False	32,768	0.5	16	1-1e-12	5.78511e-08
lomax	moment	True	39,913.1	1	16	1-1e-12	—
	sbj	False	271,476	5	16	E[N]-adj 1-1e-12	—
	used	False	65,536	1	16	1-1e-12	4.95711e-08

All three take the `moment` window. The single big jump reach of the gamma is twice its window and is not used, because the tail is exponential. For the lognormal and the Lomax the reach is six and seven times the window, too far to fit in 2^{16} buckets of the same size, so the window stays and the clipped probability, about 5×10^{-8}, is reported. The one-line `bs_description` gives the window and the `log2` that would hold the reach at the same bucket, and `validation_description` gives the result of the moment check.

```
for k, a in books.items():
    wprint(f'{k}: {a.validation_description}')
    wprint(' ', a.bs_description)
```

```
gamma: not unreasonable
  moment grid: bs=1/4, log2=16, window [0, 16,384] of width 16,384
lognorm: not unreasonable
  moment grid: bs=1/2, log2=16, window [0, 32,768] of width 32,768; clips
  ~5.79e-08 of the tail (raise log2 to 19)
lomax: fails sev skew, agg skew
  moment grid: bs=1, log2=16, window [0, 65,536] of width 65,536; clips
  ~4.96e-08 of the tail (raise log2 to 19)
```

The Lomax fails the skewness check. Its third moment is finite, but the part beyond x, $\mathsf{P}(X^3; X > x)$, decays like $x^{3-\alpha} = x^{-0.6}$, so beyond the top of the window, 65,536, it is still about 9 percent of the whole.

When the reach does fit, the window grows to it. A lognormal with CV 0.75 has a reach that fits at the same bucket, so the `moment` row is floored up to the `sbj` row: the two share `x_max`, and the `note` column ends `sbj floor`.

```
b = build('agg Lognorm2 20 claims sev lognorm 100 cv 0.75 poisson')
GT(b.bs_window_df[['x_max', 'W', 'bs', 'log2', 'note']], float_format=',.6g')
```

method	x_max	W	bs	log2	note
moment	13,433	13,433	0.25	16	3-moment MoM window; sbj floor
sbj	13,433	13,433	0.25	16	single big jump: ES - mu_X + q_X(p**)
used	16,384	16,384	0.25	16	realized grid (moment)

Capturing a clipped tail

The `log2` suggested by `bs_description`, 19 for the lognormal, assumes the same bucket. Passing `log2=19` alone makes the sizer refine the bucket to 1/16 over the same window, and the clip remains. Fixing the bucket as well, with `bs=1/2`, extends the window to 262,144, past the reach of 180,738.

```
for kw in [dict(log2=19), dict(bs=1/2, log2=19)]:
    wprint(kw, build(programs['lognorm'], **kw).bs_description)
```

```
{'log2': 19} moment grid: bs=1/16, log2=19, window [0, 32,768] of width 32,768;
clips ~5.79e-08 of the tail (raise log2 to 22)
{'bs': 0.5, 'log2': 19} moment grid: bs=1/2, log2=19, window [0, 262,144] of
width 262,144
```

Return periods for both tails

The return period table `tail_df` is indexed by the non-exceedance probability P and is symmetric about the median: each period T gives a row at $1/T$ and one at $1 - 1/T$. The method `tail_periods_df(periods)` computes the table for another list of periods. The columns `VaR` and `TVaR` are the value at risk (the quantile) and the tail value at risk, `xsVaR` is `VaR` less the mean, and `VaR/Mean` is the ratio of `VaR` to the mean.

```
GT(books['lognorm'].tail_periods_df([2, 10, 100, 1000]),
   formatters={'VaR': 0, 'TVaR': 0, 'xsVaR': 0, 'VaR/Mean': 3})
```

P	T	VaR	TVaR	xsVaR	VaR/Mean
0.0010	1,000	426	2,002	-1,574	0.213
0.0100	100	666	2,015	-1,334	0.333
0.1000	10	1,107	2,122	-893	0.554
0.5000	2	1,880	2,606	-120	0.940
0.9000	10	3,028	3,671	1,028	1.514
0.9900	100	4,488	5,271	2,488	2.244
0.9990	1,000	6,294	7,467	4,294	3.147

Using the structure chart

Reinsurance program diagrams of layers, shares and retentions, with optional Lee curves (the structure chart)

Stephen J. Mildenhall

2026-11-01

The structure chart draws a reinsurance program the way a broker's slide does. A gross block on the left shows the subject book. Beside it, each cession stage is a tower over its loss axis: one for the occurrence program, with loss per claim, and one for the aggregate program, with annual loss. Each placed layer is a rectangle from its attachment to its exhaustion point, and its **width is its share**. The unplaced part of a layer is hatched (co-participation), retentions are gray, and an uncovered band is drawn as a gap with a cross hatch. The loss axis of a tower is ticked at its layer boundaries rather than on a continuous scale.

Each block is annotated with fields selected by the `annotate` argument from `geometry`, `premium`, `el` (expected loss), `lr` (loss ratio), `rol` (rate on line), `lol` (loss on line), `sd`, `pr_attach`, `pr_detach`, `reinstatements`, and `cede`, always in that order. The default is `('geometry', 'premium', 'el', 'lr')`. A field is available at one of three **tiers**, and a field whose tier is not reached is omitted, so one selection works on any object.

1. *Declared*: the terms and name of each layer, available before the aggregate is updated.
2. *Built*: the risk statistics from `reins_stats_df` (`el`, `sd`, `lol`, `pr_attach`, `pr_detach`), on an updated aggregate.
3. *Priced*: `premium`, `lr`, `rol`, `cede`, and `reinstatements`, on a `pnl` or `xpnl` only, because a plain `agg` drops pricing clauses (see Using pnl and xpnl).

Premium, expected loss, and standard deviation are annual and shown at 100% terms, as a layer is quoted. With `lee=True`, the chart of an updated object also shows, beside each tower and on the same loss axis, the quantile function (Lee curve) of the distribution that tower is read against: the gross claim severity for the occurrence tower, and for the aggregate tower the annual loss after the occurrence program, which is the subject of the aggregate cover. Faint lines extend each boundary across to the curve.

The function `build_chart_doc` returns the chart as a `ChartDoc`, which contains the blocks and curves as data. The function `plot_chartdoc` draws it with matplotlib. The module `aggregate.charts` is provisional: its interface may change in a minor release.

```
doc = build_chart_doc(obj, 'structure', annotate=..., lee=False)  # obj: Aggregate or PnL
fig = plot_chartdoc(doc, return_period=False, log=False)
```

```
import numpy as np
from aggregate import build, __version__
from aggregate.charts import available_charts, build_chart_doc
from aggregate.plots import plot_chartdoc
from style import wprint, refuse, figures
```

```
figures()
print(__version__)
```

```
1.54.0
```

The declared and built tiers on an aggregate

The example book has 20 claims a year from a lognormal severity with mean 100 and coefficient of variation 3, limited to 2,000, and an occurrence program of three named layers: `Working`, 250 xs 250 in full, `Middle`, 90% of 500 xs 500, and `Top`, 75% of 1,000 xs 1,000. The structure chart is available as soon as the program is declared. The function `available_charts` lists it with the other charts of the object.

```
book = '''agg Book 20 claims 2000 xs 0 sev lognorm 100 cv 3
  occurrence net of
    250 xs 250 as "Working" and
    90% po 500 xs 500 as "Middle" and
    75% po 1000 xs 1000 as "Top"
  poisson'''
u, a = build(book, update=False), build(book)
print(available_charts(u))
```

```
['structure', 'reins']
```

The same selection of fields gives only the terms on the declared object and adds the statistics on the built one. The tiers reached are recorded in the document's `meta`. The expected loss of `Middle` is 198.8 at 100%, so the expected loss of the 90% placed share is $0.9 \times 198.8 = 178.9$, the mean of `layer.2` in `reins_stats_df`. Loss on line is the same on either basis.

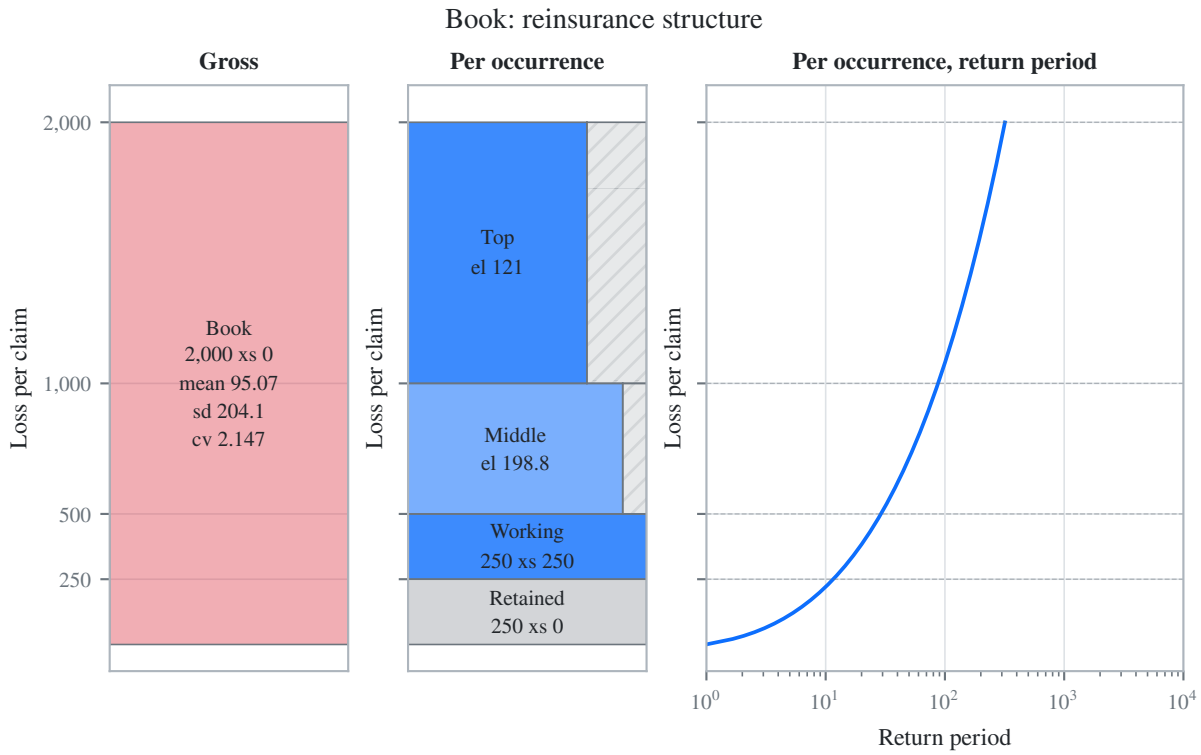
```
fields = ('geometry', 'el', 'sd', 'lol', 'pr_attach', 'pr_detach')
for obj in [u, a]:
    doc = build_chart_doc(obj, 'structure', annotate=fields)
    print(doc.meta['tiers'])
    for b in doc.blocks:
        if b.role == 'layer':
            wprint(f'  {b.label:8s}', ', '.join(b.label_lines))
print(f"{a.reins_stats_df.loc[('agg', 'mean'), ('occ', 'layer.2')]:.1f}")
```

```
['declared']
  Working  250 xs 250
  Middle   90.0% po 500 xs 500
  Top      75.0% po 1,000 xs 1,000
['declared', 'built']
  Working  250 xs 250, el 272.1, lol 108.8%, sd 240.1, pr attach 8.6%, pr detach
           3.4%
  Middle   90.0% po 500 xs 500, el 198.8, lol 39.8%, sd 285.1, pr attach 3.4%,
           pr detach 1.1%
  Top      75.0% po 1,000 xs 1,000, el 121, lol 12.1%, sd 308.8, pr attach 1.1%,
           pr detach 0.3%
178.9
```

Lee curves and return periods

With `lee=True` each tower is joined to its Lee curve. Drawn with `return_period=True` and a log scale on the horizontal axis, the curve gives the return period of every boundary. The renderer drops any annotation line that does not fit inside its block: the partial layers `Middle` and `Top` are too narrow for their terms, and the short `Working` block has no room for its expected loss.

```
fig = plot_chartdoc(build_chart_doc(a, 'structure', lee=True),
                    return_period=True, log='x')
```



On the occurrence tower the curve is the quantile function of the claim severity, so its return period counts **claims, not years**. The attachment of `Working` is exceeded by one claim in 11.6, and with 20 claims a year it is exceeded in 82% of years, an annual return period of 1.2 years. The annual figures assume a Poisson count, and the conversion is described in Using `oep`. The aggregate tower's curve is an annual distribution, and its return period is in years.

```
for x in [250, 500, 1000]:
    s = a.sev.sf(x)
    annual = -np.expm1(-a.n * s)
    print(f'{x:5d}  {1 / s:5.1f} claims  {annual:.4f}  {1 / annual:4.2f} years')
```

```
  250   11.6 claims  0.8228  1.22 years
  500   29.0 claims  0.4977  2.01 years
 1000   87.6 claims  0.2042  4.90 years
```

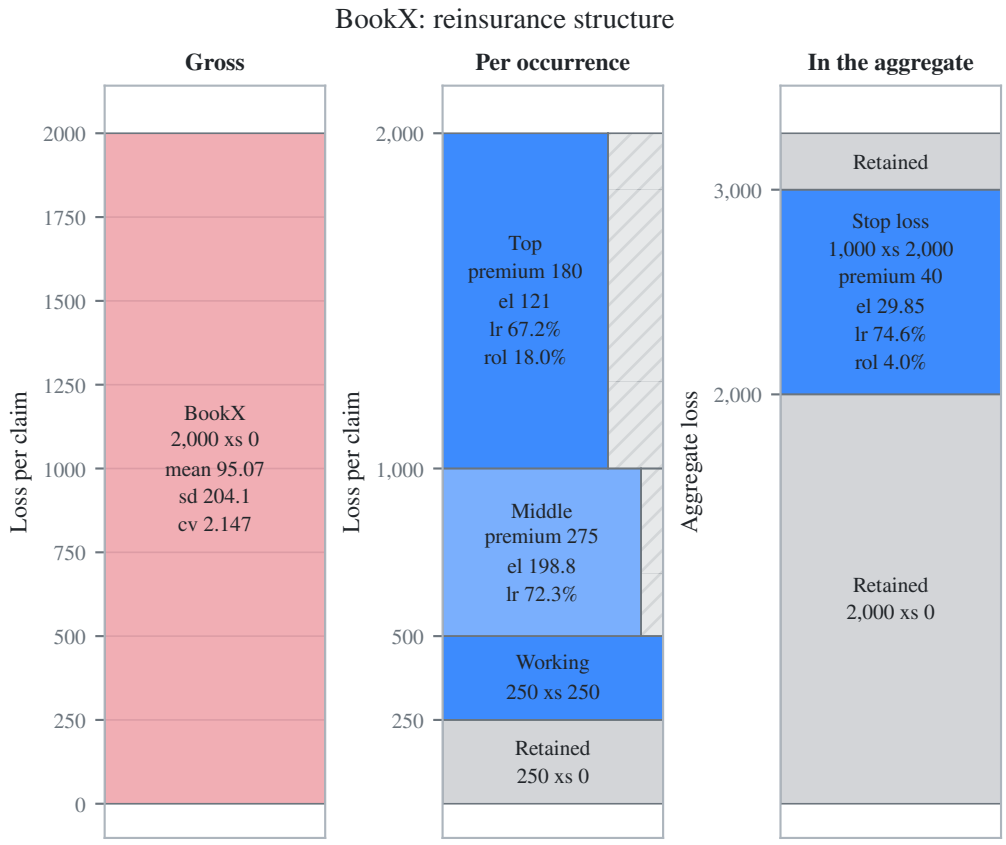
The priced tier on an xpnl

The `xpnl` below wraps the example book with a premium on each layer and adds an aggregate stop loss, 1,000 xs 2,000, on the annual net of the occurrence program. Every field is now available. Premium is shown at 100% terms: the `Middle` layer is rated at 55% on line, a premium of 275 for the full

layer, and its loss ratio is 198.8/275 = 72.3%. The placed premiums, 375, 247.5, and 135, are listed in the `economics` of the `PnL`. The aggregate tower is read against the annual net loss, so its statistics are annual.

```
x = build('''xpnl BookX 2500 premium less
  agg BookX 20 claims 2000 xs 0 sev lognorm 100 cv 3
    occurrence net of
      250 xs 250 rol 150% as "Working" and
      90% po 500 xs 500 rol 55% cede 10% as "Middle" and
      75% po 1000 xs 1000 rol 18% as "Top"
    poisson
    aggregate net of 1000 xs 2000 rol 4% as "Stop loss"''')
doc = build_chart_doc(x, 'structure',
                      annotate=('geometry', 'premium', 'el', 'lr', 'rol', 'cede'))
print(doc.meta['tiers'], [round(v, 2) for v in x.economics['pc_occ_by_layer']])
fig = plot_chartdoc(doc)
```

```
['declared', 'built', 'priced'] [375.0, 247.5, 135.0]
```



The `Middle` block shows three of its six annotation lines, losing its terms for width and its rate on line and commission for height. The full annotation of every layer is in the document.

```
for b in doc.blocks:
    if b.role == 'layer':
        wprint(f'{b.label:9s}', ', '.join(b.label_lines))
```

```
Working   250 xs 250, premium 375, el 272.1, lr 72.6%, rol 150.0%
Middle    90.0% po 500 xs 500, premium 275, el 198.8, lr 72.3%, rol 55.0%, cede
          10.0%
Top       75.0% po 1,000 xs 1,000, premium 180, el 121, lr 67.2%, rol 18.0%
Stop loss 1,000 xs 2,000, premium 40, el 29.85, lr 74.6%, rol 4.0%
```

Refused requests

A Lee curve needs an updated object, an `annotate` field must be one of those listed in the error message, and an aggregate with no reinsurance has no structure chart. Each request raises a `ValueError`.

```
plain = build('agg Plain 20 claims 2000 xs 0 sev lognorm 100 cv 3 poisson')
refuse(lambda: build_chart_doc(u, 'structure', lee=True),
       lambda: build_chart_doc(a, 'structure', annotate=('el', 'var')),
       lambda: build_chart_doc(plain, 'structure'))
```

```
chart_structure: lee=True needs the realized grid: call Book.update() first, or
take the geometry-only chart the default gives.
unknown annotate field(s) ['var']; expected a subset of ('geometry', 'premium',
'el', 'lr', 'rol', 'lol', 'sd', 'pr_attach', 'pr_detach', 'reinstatements',
'cede')
chart 'structure' is not available for Aggregate 'Plain': its availability
predicate failed (missing update() grid, or required structure such as
reinsurance)
```

Using `tsallis`

The Tsallis distortion, a Beta mixture of TVaRs with a closed-form Gini index, and the extended calibration set

Stephen J. Mildenhall

2026-11-06

The **Tsallis distortion**[1] of order $\alpha > 0$ is the mixture of the TVaR distortions $\min(s/t, 1)$ over the tail probability t, weighted by the Beta$(\alpha, 1)$ density $\alpha t^{\alpha-1}$:

$$g_\alpha(s) = \int_0^1 \min(s/t, 1)\, \alpha t^{\alpha-1}\, dt = \frac{\alpha s - s^\alpha}{\alpha - 1},$$

with $g_1(s) = s(1 - \log s)$ the limit at $\alpha = 1$. It has no jump at zero, and it puts no weight on the mean or the maximum alone. Its Gini index, $2\int_0^1 g - 1$ (see Using distortions), is $1/(\alpha+1)$, so the member with index G is $\alpha = 1/G - 1$. The family runs from the maximum as $\alpha \to 0$ to the mean as $\alpha \to \infty$, and $\alpha = 2$ is the dual distortion $1 - (1 - s)^2$.

The name comes from the price. Since $g_\alpha(S) = S + (S - S^\alpha)/(\alpha - 1)$,

$$g_\alpha[X] = \mathsf{P}X + \int_0^\infty \frac{S(x) - S(x)^\alpha}{\alpha - 1}\, dx,$$

the expected loss plus the cumulative residual Tsallis entropy of X of order α. At $\alpha = 1$ the loading is the cumulative residual entropy $-\int_0^\infty S \log S\, dx$. The family is used as a sixth pricing family beside the five standard ones. Like them it has one parameter and no jump at zero, and its slope at zero is $\alpha/(\alpha - 1)$ for $\alpha > 1$, finite where PH and Wang have an infinite slope.

```
distortion NAME [as "label"] tsallis alpha
distortion T tsallis 3                          # index 1/4
distortion Half as "Tsallis half" tsallis 0.5   # g(s) = 2 sqrt(s) - s, index 2/3
```

```
import numpy as np
import pandas as pd
from aggregate import build, Distortion, __version__
from aggregate.spectral import consistent_distortions
from style import wprint, GT, refuse, figures, figure
figures()
print(__version__)
```

```
2.1.0a5
```

[1] From aggregate 2.1.0a3.

Tsallis distortions from DecL and from Python

The DecL statement `distortion` with the kind `tsallis` builds a Tsallis distortion. In Python the factory `Distortion('tsallis', alpha)`, the static method `Distortion.tsallis` and the constructor `Distortion.from_gini('tsallis', 0.25)` build the same object. All four below have the label `Tsallis(3)`, the index 0.25 and the same `id`.

```
t = build('distortion T tsallis 3')
for d in [t, Distortion('tsallis', alpha=3.0), Distortion.tsallis(3.0),
          Distortion.from_gini('tsallis', 0.25)]:
    print(f'{type(d).__name__:18s} {d.label:10s} {d.gini:.4f} {d.id()}')
```

```
TsallisDistortion  Tsallis(3) 0.2500 912DAE9A
TsallisDistortion  Tsallis(3) 0.2500 912DAE9A
TsallisDistortion  Tsallis(3) 0.2500 912DAE9A
TsallisDistortion  Tsallis(3) 0.2500 912DAE9A
```

The attribute `info` summarizes the distortion. Its `weights mean` and `weights max` rows are both 0, and its TVaR mixture has no atoms inside $(0, 1)$, since the Beta mixture is continuous.

```
wprint(t.info)
```

```
distortion object name   tsallis
kind                     tsallis
kind name                tsallis (beta(alpha, 1) mixture of tvars)
shape                    3
shape name               alpha
other params             none
weights mean             0
weights max              0
interior atoms           False
gini                     0.25
area                     0.625
id                       912DAE9A
```

The Tsallis distortion beside the standard families

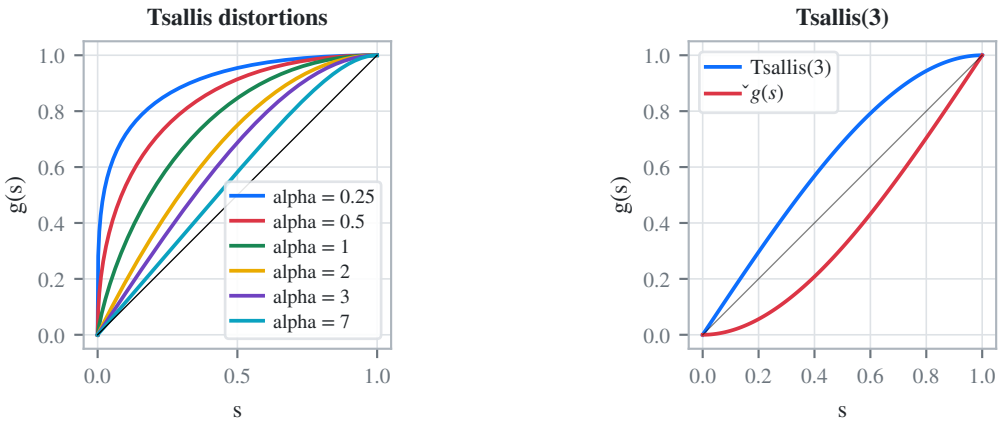
The function `consistent_distortions(p, extended=True)` in `aggregate.spectral` returns the five standard families and `tsallis`, all with index p. The table shows $g(s)$ at five values of s for index 0.25. At $s = 0.001$, an event in the extreme tail, only TVaR gives less weight than Tsallis, 0.0013 against 0.0015, and at $s = 0.5$ Tsallis gives the most, 0.6875. Of the six it is closest to dual, which gives 0.0017 at $s = 0.001$ and 0.6850 at $s = 0.5$.

```
fams = consistent_distortions(0.25, extended=True)
ss = [0.001, 0.01, 0.1, 0.5, 0.9]
GT(pd.DataFrame([[d.gini, d.label, *d.g(np.array(ss))] for d in fams.values()],
                columns=['gini', 'label', *[f'g({x})' for x in ss]],
                index=pd.Index(list(fams), name='kind')), float_format='.4f')
```

kind	gini	label	g(0.001)	g(0.01)	g(0.1)	g(0.5)	g(0.9)
ccoc	0.2500	ccoc(0.333)	0.2507	0.2575	0.3250	0.6250	0.9250
ph	0.2500	PH(0.6)	0.0158	0.0631	0.2512	0.6598	0.9387
wang	0.2500	Wang(0.451)	0.0042	0.0303	0.2030	0.6739	0.9584
tsallis	0.2500	Tsallis(3)	0.0015	0.0150	0.1495	0.6875	0.9855
dual	0.2500	dual(1.67)	0.0017	0.0166	0.1610	0.6850	0.9785
tvar	0.2500	TVaR(0.25)	0.0013	0.0133	0.1333	0.6667	1.0000

The left panel shows g_α for six orders, from $\alpha = 0.25$, index 0.8, to $\alpha = 7$, index 0.125. The right panel is `Distortion.plot` for $\alpha = 3$, which draws g and its dual $\check{g}$.

```
s = np.linspace(0, 1, 1001)
fig, (ax0, ax1) = figure('full', cols=2)
for alpha in [0.25, 0.5, 1, 2, 3, 7]:
    ax0.plot(s, Distortion.tsallis(alpha).g(s), label=f'alpha = {alpha}')
ax0.plot([0, 1], [0, 1], c='k', lw=0.5)
ax0.set(xlabel='s', ylabel='g(s)', aspect='equal', title='Tsallis distortions')
ax0.legend(fontsize='small')
t.plot(ax=ax1);
```



Calibrating the extended set

The method `calibrate_distortions` of an aggregate or a portfolio calibrates the five standard families by default, and with `extended=True` it adds `tsallis` between `wang` and `dual`. The table shows the six families fitted to a 10% cost of capital at the 99% VaR of a book with 50 expected claims and a lognormal severity with mean 40 and CV 1.8 (see Using `price` and `evaluate` for the reverse question). The Tsallis order is solved for in its index, and the fitted $\alpha = 4.475$ has index 0.183, between dual at 0.175 and TVaR at 0.185. The five standard rows are the same as a default calibration gives.

```
book = build('agg Book 50 claims sev lognorm 40 cv 1.8 poisson')
res = book.calibrate_distortions(0.1, p=0.99, extended=True)
GT(res.distortion_df[['gini', 'param_name', 'param', 'error']],
   formatters={'gini': '{x:.4f}', 'param': '{x:.4g}'})
```

distortion	gini	param_name	param	error
ccoc	0.0909	r	0.1	0.00e+00
ph	0.1392	a	0.7555	8.88e-10
wang	0.1581	lam	0.2821	2.60e-07
tsallis	0.1826	alpha	4.475	0.00e+00
dual	0.1752	b	1.425	-8.71e-09
tvar	0.1853	p	0.1853	7.44e-06

The price at order one and the cumulative residual entropy

The price of the book under g_1 is its expected loss plus the cumulative residual entropy $-\int S \log S \, dx$. Both sides, computed from the survival function of the book on its grid, agree to rounding:

```
S = book.density_df.S.to_numpy()
el = S.sum() * book.bs
cre = -(S[S > 0] * np.log(S[S > 0])).sum() * book.bs
print(f'g_1[X] = {Distortion.tsallis(1).g(S).sum() * book.bs:,.4f}, '
      f'P X + CRE = {el:,.4f} + {cre:,.4f} = {el + cre:,.4f}')
```

```
g_1[X] = 2,566.8986, P X + CRE = 1,999.9998 + 566.8988 = 2,566.8986
```

Refused calls

Four calls are refused: an order that is not positive, a Tsallis distortion with two parameters in DecL, `calibrate_distortions` with both `names` and `extended=True`, which choose the same thing, and `Distortion.from_gini` with an index outside $(0, 1)$, which no Tsallis distortion reaches.

```
refuse(lambda: Distortion('tsallis', alpha=0),
       'distortion T tsallis 3 4',
       lambda: book.calibrate_distortions(0.1, p=0.99, names=['ph'], extended=True),
       lambda: Distortion.from_gini('tsallis', 1.0))
```

```
TsallisDistortion: alpha must be finite and positive, got 0
DecL distortion 'tsallis' expects 1 parameter(s) ('alpha',), got 2: [3.0, 4.0]
calibrate_distortions takes names= or extended=True, not both: extended=True is
the families ('ccoc', 'ph', 'wang', 'tsallis', 'dual', 'tvar').
from_gini needs a Gini index in the open interval (0, 1), got 1.0.
```

Using `tweedie`

Tweedie parameter conversions and named laws with the tweedie statement

Stephen J. Mildenhall

2026-10-30

A Tweedie distribution with power variance function $V(\mu) = \mu^p$, $1 < p < 2$, has two common parameterizations. A GLM uses the reproductive form $\mathrm{Tw}_p(\mu, \phi)$, with mean μ, dispersion ϕ and variance $\phi\mu^p$. An actuary uses the compound Poisson form: an expected claim count λ and a gamma severity with mean m and coefficient of variation (CV) ν_X. The two are linked by

$$\lambda = \frac{\mu^{2-p}}{(2-p)\phi}, \qquad \alpha = \frac{2-p}{p-1}, \qquad m = \mu/\lambda, \qquad \nu_X = \alpha^{-1/2},$$

where α is the gamma shape. The Tweedie CV is $\nu = \sqrt{\phi}\,\mu^{p/2-1}$, and the probability of no loss is $e^{-\lambda}$.

The DecL statement `tweedie`[1] converts between these parameterizations. It accepts three forms and returns a `TweedieConversion`. It is used to find the claim count and severity implied by a fitted Tweedie GLM. Without a name the statement stores nothing. With a name, as in `tweedie t0 1.5 100 2`, it stores the law in the recipe base, and the reference `tweedie.t0` returns it, as described in Using `stable` names and arithmetic.

```
tweedie 1.5 100 2                       # p, mean, dispersion
tweedie 1.5 100 cv 0.447214             # p, mean, CV
tweedie 10 claims sev gamma 10 cv 1     # Poisson count and gamma severity
tweedie t0 1.5 100 2                    # the first law, stored as tweedie.t0
```

```
from aggregate import build, qd, TweedieConversion, __version__
from style import wprint, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

Converting between Tweedie parameterizations

Each of the three forms of the `tweedie` statement returns a `TweedieConversion`. Printed, it shows the distribution in all three forms, joined by =, followed by the probability of no loss, to six significant figures. Each form is valid DecL and can be pasted back into a program.

[1] From aggregate 1.22.0

```
t = build('tweedie 1.5 100 2')
wprint(t)
```

```
tweedie 1.5 100 2 = tweedie 1.5 100 cv 0.447214 = tweedie 10 claims sev gamma 10
cv 1, Pr(X=0) 4.53999e-05
```

A `TweedieConversion` is a named tuple of full-precision floats. Its fields are available for further calculation, and are not rounded as the printed form is.

```
t
```

```
TweedieConversion(p=1.5, mean=100.0, dispersion=2.0, cv=0.447213595499958, frequency=10.0,
sev_m=10.0, sev_cv=1.0, p0=4.5399929762484854e-05)
```

The four programs below describe the same distribution, and each returns an expected claim count of 10. The compound form parses its severity with the same `sev` clause as an `agg`, so any gamma severity an `agg` accepts is accepted here, including `sev 10 * gamma 1`. The CV form reproduces the parameters only to the six figures given, and its count is 9.99998.

```
for program in ['tweedie 1.5 100 2',
                'tweedie 1.5 100 cv 0.447214',
                'tweedie 10 claims sev gamma 10 cv 1',
                'tweedie 10 claims sev 10 * gamma 1']:
    print(f'{program:40s}', build(program).frequency)
```

```
tweedie 1.5 100 2                        10.0
tweedie 1.5 100 cv 0.447214              9.999981910232723
tweedie 10 claims sev gamma 10 cv 1      10.0
tweedie 10 claims sev 10 * gamma 1       10.0
```

Frequency and severity implied by a Tweedie GLM

A fitted Tweedie GLM gives the power p and the dispersion ϕ, and the linear predictor gives the mean μ for each risk. The `tweedie` statement converts these to the expected claim count and gamma severity they imply, shown here for three risks.

```
for mean in [50, 250, 1000]:
    wprint(build(f'tweedie 1.65 {mean} 30'))
```

```
tweedie 1.65 50 30 = tweedie 1.65 50 cv 2.76211 = tweedie 0.374499 claims sev
gamma 133.512 cv 1.36277, Pr(X=0) 0.687634
tweedie 1.65 250 30 = tweedie 1.65 250 cv 2.08411 = tweedie 0.657794 claims sev
gamma 380.058 cv 1.36277, Pr(X=0) 0.517993
tweedie 1.65 1000 30 = tweedie 1.65 1000 cv 1.63516 = tweedie 1.06859 claims sev
gamma 935.813 cv 1.36277, Pr(X=0) 0.343493
```

With p and ϕ fixed, the severity CV depends on p alone, $\nu_X^2 = (p-1)/(2-p)$. A larger mean comes from both more claims and larger claims, with $\lambda \propto \mu^{2-p}$ and $m \propto \mu^{p-1}$.

The tweedie clause in an aggregate

Inside an `agg` program, the `tweedie` clause takes the dispersion form or the `cv` form. The stored program keeps the form as written, as `pprogram` shows.

```
a = build('agg TW tweedie 1.5 100 cv 0.447214')
qd(a)
print(a.pprogram)
```

```
Aggregate object: TW. Frequency distribution poisson. Severity gamma,
[0, inf), exponential right tail. Updated with bucket size 1/64 and log2
= 16. Validation: not unreasonable.

      Mean     SD      CV    Skew     P01 Median    P99
X
Freq    10 3.1623 0.31623 0.31623
Sev     10     10       1       2 0.09375 6.9375 46.047
Agg    100 44.721 0.44721 0.67082  19.594 94.953 224.94
agg TW
  tweedie 1.5 100 cv 0.447214
```

The aggregate is computed as a compound Poisson gamma, so its variance is $\phi\mu^p$ and its probability of no loss is $e^{-\lambda}$. The first bucket is slightly larger than $e^{-\lambda}$, because it also includes the gamma probability below half a bucket.

```
t = build('tweedie 1.5 100 cv 0.447214')
print(a.density_df.p_total.iloc[0], t.p0)
print(a.est_sd ** 2, t.dispersion * t.mean ** t.p)
```

```
4.575669265299114e-05 4.540075104407702e-05
2000.0038214101514 2000.00361796
```

The method `Aggregate.as_tweedie()` returns the reproductive parameters (p, μ, ϕ) of any compound Poisson gamma aggregate, however it was specified. The constructor `TweedieConversion.from_dispersion` converts them to the full conversion.

```
b = build('agg Longhand 10 claims sev gamma 10 cv 1 poisson')
print(b.as_tweedie())
wprint(TweedieConversion.from_dispersion(*b.as_tweedie()))
```

```
TweedieParameters(p=1.5, mean=100.0, dispersion=2.0)
tweedie 1.5 100 2 = tweedie 1.5 100 cv 0.447214 = tweedie 10 claims sev gamma 10
cv 1, Pr(X=0) 4.53999e-05
```

The TweedieConversion constructors

The class `TweedieConversion` has one constructor for each form of the statement: `from_dispersion`, `from_cv`, and `from_compound`. Each returns the same object as the corresponding DecL statement, so the conversion is available in Python without writing a program.

```
print(TweedieConversion.from_cv(1.5, 100, 0.447214) == build('tweedie 1.5 100 cv 0.447214'))
wprint(TweedieConversion.from_compound(10, 10, 1))
```

```
True
tweedie 1.5 100 2 = tweedie 1.5 100 cv 0.447214 = tweedie 10 claims sev gamma 10
cv 1, Pr(X=0) 4.53999e-05
```

Refused programs

A power outside $1 < p < 2$ is refused, because it has no compound Poisson form, and so is a compound form whose severity is not an unshifted gamma, such as a lognormal or a shifted gamma. Each of these raises a `ValueError` that identifies the form. The other powers are available through a `chfagg`, as described in Using `AggregateCHF`. An `agg` takes the `tweedie` clause but not a reference such as `tweedie.t0`.

```
refuse('tweedie 2.5 100 2',
       'tweedie 10 claims sev lognorm 10 cv 1',
       'tweedie 10 claims sev gamma 10 cv 1 + 5',
       'agg A tweedie.t0')
```

```
tweedie: p must be strictly between 1 and 2, got 2.5. The clause is `tweedie <p>
<mean> <dispersion>` or `tweedie <p> <mean> cv <cv>`, shape parameter first. (It
read `<mean> <p> <dispersion>` before 1.0.0a231, so an older program needs its
first two numbers swapped.)
tweedie: the compound form needs a plain gamma severity, `tweedie <count> claims
sev gamma <mean> cv <cv>`, but found severity 'lognorm'.
tweedie: the compound form needs a plain gamma severity, `tweedie <count> claims
sev gamma <mean> cv <cv>`, but found sev_loc = 5.0.
DecL parse error at line 1, column 7: Unexpected 'tweedie.t0'. Did you mean:
tweedie?
```

Using unconditional layers

Conditional and unconditional layer severities, and the trailing ! on a severity

Stephen J. Mildenhall

2026-11-02

A layer y xs a pays $g(X) = \min((X-a)^+, y)$ on a ground-up loss X. It has two natural severities. The **conditional** severity is the law of $g(X)$ given $X > a$, the payment on a loss that reaches the layer. The **unconditional** severity is the law of $g(X)$ itself, which includes a zero payment for each loss that does not reach it. The two are related by

$$\Pr(g(X) \le t) = \Pr(X \le a) + \Pr(X > a)\ \Pr(g(X) \le t \mid X > a), \quad t \ge 0,$$

so the unconditional severity is the conditional one with an extra mass $\Pr(X \le a)$ at zero, and $\mathsf{P}g(X) = \Pr(X > a)\,\mathsf{P}[g(X) \mid X > a]$. The conditioning is strict: a loss of exactly a pays nothing and is excluded.

In DecL the layers clause `y xs a` sits between the exposure and the severity. By default the severity is conditional, and the claim count in the exposure clause is the count of losses that reach the layer. A trailing `!` after the severity makes it unconditional: the claim count is the ground-up count, and each loss below the attachment adds a zero payment. It is used when the claim count is a ground-up count, from a ground-up frequency study or one claim per policy, and the payment to a layer is needed. A program with no layers clause uses the ground-up severity as it stands, including any mass at zero, and conditions on nothing. The grammar accepts `!` after any `sev` or `dsev`, but without a layers clause it has no effect. The same is true with a layers clause when $\Pr(X > a) = 1$.

```
agg Eg dfreq [1] dsev [0:4]             # ground up, no layers clause
agg Eg dfreq [1] dsev [0:4] !           # the same, nothing to condition on
agg Eg dfreq [1] 3 xs 0 dsev [0:4]      # min(X, 3) given X > 0
agg Eg dfreq [1] 3 xs 0 dsev [0:4] !    # min(X, 3)
agg Eg dfreq [1] 2 xs 2 dsev [0:4]      # min((X - 2)+, 2) given X > 2
agg Eg dfreq [1] 2 xs 2 dsev [0:4] !    # min((X - 2)+, 2)
```

```
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.63.2
```

Conditional and unconditional densities of one claim

The six programs have one claim for sure, `dfreq [1]`, with a ground-up loss X equally likely to be 0, 1, 2, 3, or 4. With one claim the aggregate is the layer severity, so the column `p_total` of `density_df` is the density of the payment. The table shows the density under each program, one column per program, grouped by layers clause with the default form beside the form with `!`.

```
aggs = {}
for layer in ['', '3 xs 0', '2 xs 2']:
    for bang in ['', '!']:
        key = (layer or 'no layer', bang or 'default')
        aggs[key] = build(f'agg Eg dfreq [1] {layer} dsev [0:4] {bang}')
density = pd.DataFrame({k: a.density_df.p_total for k, a in aggs.items()})
GT(density.loc[0:4].fillna(0).rename_axis('payment'))
```

	no layer		3 xs 0		2 xs 2	
payment	default	!	default	!	default	!
0	0.2000	0.2000	0.0000	0.2000	0.0000	0.6000
1	0.2000	0.2000	0.2500	0.2000	0.5000	0.2000
2	0.2000	0.2000	0.2500	0.2000	0.5000	0.2000
3	0.2000	0.2000	0.5000	0.4000	0.0000	0.0000
4	0.2000	0.2000	0.0000	0.0000	0.0000	0.0000

With no layers clause the payment is the loss, with probability 1/5 on each outcome including zero, and `!` changes nothing. The layer 3 xs 0 pays $\min(X, 3)$. Conditioning on $X > 0$ removes the zero loss and scales the other four up by $1/\Pr(X > 0) = 5/4$, giving 1/4 at 1 and at 2, and 1/2 at the limit 3 from the losses of 3 and 4. With `!` the zero loss stays, and the losses of 3 and 4 put 2/5 at the limit. The layer 2 xs 2 pays nothing on losses of 0, 1, and 2. Conditioning on $X > 2$ leaves the losses of 3 and 4, which pay 1 and 2 with probability 1/2 each. With `!` the payment is zero with probability 3/5, the three losses at or below the attachment, and 1 or 2 with probability 1/5 each. In both layers the unconditional column is $\Pr(X \le a)$ at zero plus $\Pr(X > a)$ times the conditional column.

A layers clause with attachment zero is therefore different from no layers clause, because it conditions on $X > 0$. The layer `inf xs 0` keeps every loss but removes the zero, so it puts 1/4 on each of 1, 2, 3, and 4, with mean 5/2 in place of the ground-up mean 2.

```
a = build('agg Eg dfreq [1] inf xs 0 dsev [0:4]')
print(a.density_df.p_total.loc[0:4].values, a.est_m)
```

```
[0.   0.25 0.25 0.25 0.25] 2.5
```

Layer means and the probability of attaching

The attribute `pattach` of a severity is $\Pr(X > a)$, here 4/5 for 3 xs 0 and 2/5 for 2 xs 2. The conditional layer means are $(1 + 2 + 3 + 3)/4 = 9/4$ and $(1 + 2)/2 = 3/2$. Multiplying by `pattach` gives the unconditional means 9/5 and 3/5, the means of the `!` programs.

```
rows = {}
for layer in ['3 xs 0', '2 xs 2']:
    c, u = aggs[(layer, 'default')], aggs[(layer, '!')]
    p = c.sevs[0].pattach
    rows[layer] = [p, c.est_m, p * c.est_m, u.est_m]
GT(pd.DataFrame.from_dict(rows, orient='index', columns=[
    'pattach', 'conditional mean', 'pattach x conditional',
    'unconditional mean']).rename_axis('layer'))
```

layer	pattach	conditional mean	pattach x conditional	unconditional mean
3 xs 0	0.8000	2.2500	1.8000	1.8000
2 xs 2	0.4000	1.5000	0.6000	0.6000

Ground-up and layer claim counts

The `!` changes the meaning of the claim count in the exposure clause, from a count of losses to the layer to a count of ground-up losses. A count of ground-up losses can always be converted into a count of losses to the layer. Each ground-up loss reaches 2 xs 2 with probability 2/5. The single ground-up claim of `dfreq [1]` therefore gives a Bernoulli count to the layer with mean 2/5, and a Poisson count of 10 ground-up claims thins to a Poisson count of 4 claims to the layer. Each pair of programs below has the same aggregate distribution, the first unconditional and the second conditional; the maximum difference in `p_total` is at the level of rounding.

```
pairs = [('agg U dfreq [1] 2 xs 2 dsev [0:4] !',
          'agg C 0.4 claims 2 xs 2 dsev [0:4] bernoulli'),
         ('agg U 10 claims 2 xs 2 dsev [0:4] ! poisson',
          'agg C 4 claims 2 xs 2 dsev [0:4] poisson')]
for pu, pc in pairs:
    u, c = build(pu, bs=1, log2=8), build(pc, bs=1, log2=8)
    diff = (u.density_df.p_total - c.density_df.p_total).abs().max()
    print(f'mean {u.est_m:.4f} {c.est_m:.4f}   max difference {diff:.2e}')
```

```
mean 0.6000 0.6000   max difference 5.55e-17
mean 6.0000 6.0000   max difference 5.55e-17
```

The two forms differ in which count is declared. With `!` the exposure clause holds the ground-up count, and the count to the layer follows from the severity. A layer moved to a new attachment then keeps the same exposure clause, where the conditional form needs a new claim count for each attachment. The deductible forms of the layers clause are on Using deductibles. The `!` after `zt` or `zm` in the frequency clause is unrelated: it sets the base mean of a modified count, and is on Using `zm` and `zt`.

Using validation

The validation verdict on a computed object, reinsured views, and the warning a failing build emits

Stephen J. Mildenhall

2026-11-01

A distribution computed on a grid can be wrong in ways that are hard to see. A bucket that misses the atoms of a severity moves probability between outcomes, and the mean can still come out right. The **validation verdict** compares what the grid holds with what the model implies and reports the result in one line. It is used to decide whether to rely on a computed object and, if not, what to change.

The verdict compares the mean, coefficient of variation (CV), and skewness of the severity and of the aggregate on the grid with their analytic values. A relative error above $\epsilon =$ `validation_eps` (10^{-4} by default) fails a mean, above 10ϵ a CV, and above 100ϵ a skewness. A pass reads `not unreasonable`, because agreement of three moments cannot prove a distribution right. The attribute `valid` holds the flags, and `valid.passes` is true when nothing failed. The words are the strings `validation_description` and `validation_explanation` (Using explanations).

The examples use these programs.

```
agg L 3 claims 1500 xs 250 dsev [1000 2000 3000] fixed
agg X 1 claims dsev [1000 2000 3000] occurrence net of 1500 xs 250 fixed
agg D 100 claims dsev [999 1000 1001] [0.333333 0.222222 0.444444] fixed
agg ZeroMean 1 claim dsev [-1 1] fixed
```

```
import warnings
from aggregate import build, __version__
from aggregate.constants import ValidationWarning
from style import wprint, figures
figures()
print(__version__)
```

```
1.58.0
```

The warning from a failing build

The warning `ValidationWarning`[1] reports a build whose verdict fails, with the verdict as its text. The aggregate `L` has three claims from a discrete severity limited to 1500 xs 250, which puts the outcomes on multiples of 250. Pinned to a bucket of 1000, the outcomes fall between grid points and the mean is wrong. Left to the sizer, the bucket lands on the outcomes and the same program passes.

[1] From aggregate 1.58.0

```
program = 'agg L 3 claims 1500 xs 250 dsev [1000 2000 3000] fixed'
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter('always')
    a = build(program, bs=1000, log2=8)
    b = build(program, bs=1000, log2=8)
print(len(caught))
wprint(str(caught[0].message))
print(a.actual_m, a.est_m, a.valid.passes)
c = build(program)
print(c.bs, c.log2, c.est_m, c.validation_description)
```

```
1
L: fails sev mean, agg mean. See validation_explanation. Further occurrences
this session are suppressed; each object records its own verdict in validation.
3750.0 3000.0000000000105 False
50.0 7 3750.0 not unreasonable
```

The warning appears once in a session for a given name and failure, so a sweep of one program reports once, and each object keeps its own verdict. A `Portfolio` warns once for itself and its units. A build the library makes on the way to something else, such as a cell of the probe in Using `sharpen` or the inner object of `sev agg.NAME`, does not warn. The warning can be turned off with `silence_warnings(ValidationWarning)`.

Validating a reinsured view

An object with reinsurance reports a net or ceded view, and its verdict has two clauses. The first compares the gross subject with its analytic moments. The second compares the reported view with the exact image of the subject on the grid: the net or ceded amount of each bucket, before that amount is placed back on the grid. The difference is the error of placing it there. The word in the second clause is `net` or `ceded`, or `output` when the occurrence and aggregate covers report different views.

The aggregate `X` has one claim of 1000, 2000, or 3000, and its net outcomes under 1500 xs 250 are 250, 500, and 1500. On a bucket of 1000 the subject is exact and the net mean of 750 is right, but the net CV is 0.96 against the true 0.72. On the bucket the sizer selects, both clauses pass.

```
program = 'agg X 1 claims dsev [1000 2000 3000] occurrence net of 1500 xs 250 fixed'
with warnings.catch_warnings():
    warnings.simplefilter('ignore', ValidationWarning)
    coarse = build(program, bs=1000, log2=8)
wprint(coarse.validation_description)
print(coarse.est_m, round(coarse.est_cv, 4), coarse.valid.passes)
fine = build(program)
wprint(fine.validation_description)
print(fine.bs, fine.est_m, round(fine.est_cv, 4), fine.valid.passes)
```

```
reinsurance; subject not unreasonable; net fails sev cv, agg cv
750.0 0.9623 False
reinsurance; subject not unreasonable; net not unreasonable
50.0 750.0 0.7201 True
```

Either clause failing fails the object. A failing subject with a clean view still means the gross density is wrong, along with anything priced on it. For a continuous severity the image on the grid is the best reference available rather than the exact law, so the second clause measures only the placement.

Skewness when the mean is large against the spread

The skewness is computed from the raw moments through the third central moment, $\mathsf{P}X^3 - 3\,\mathsf{P}X\,\mathsf{P}X^2 + 2(\mathsf{P}X)^3$, which subtracts nearly equal numbers when the mean is large against the spread. The grid's skewness then carries a rounding error that no bucket removes, on an exact law as much as on a wrong one. When the third central moment is below 10^{-12} of the terms subtracted, the skewness cannot be held to the 1% the test requires, so it is reported and not tested.

The aggregate `D` is exact on its grid, with a mean of 100,000 and a CV of 9×10^{-5}. Its analytic skewness is -0.021, close to the true -0.022. The skewness on the grid is printed second and is well away from both, and it varies with the numerical library in use.

```
d = build('agg D 100 claims dsev [999 1000 1001] [0.333333 0.222222 0.444444] fixed')
st = d.stats_df.loc[('agg', 'skew')]
print(round(st['mixed'], 4), round(st['empirical'], 4), d.validation_description)
```

```
-0.0213 -0.0164 not unreasonable
```

Moments that do not exist

A CV or skewness can be `nan` because the law has no such moment: a CV at a zero mean, a skewness at a zero variance, and either one after an infinite moment. The verdict skips those comparisons. A `nan` the library failed to compute is different, from a negative variance or a `nan` raw moment. That verdict opens with `analytic cv, skew undefined`, and `passes` is false, since nothing has checked those moments.

```
z = build('agg ZeroMean 1 claim dsev [-1 1] fixed')
print(z.stats_df.loc[('agg', 'cv'), 'mixed'], z.validation_description, z.valid.passes)
```

```
nan not unreasonable True
```

A law computed from its characteristic function uses the same words, with its own checks on the inversion (Using `AggregateCHF`). On a stable law with a mean, the verdict compares the mean on the window plus its closed-form tail with the theoretical mean (Using `AggregateCHF` with stable laws).

Using `yelt`

Aggregates from year loss tables with the yelt clause

Stephen J. Mildenhall

2026-10-30

A year loss table (YELT) lists simulated years y, each with its annual loss s_y and optionally a probability p_y, otherwise $1/n$ for n years. Each row is already a whole annual outcome, so there is no compounding: the aggregate is the discrete law $\Pr(S = s) = \sum_y p_y 1\{s_y = s\}$, built as `dfreq [1] dsev` over the annual losses, with population moments $\mathsf{P}S^r = \sum_y p_y s_y^r$.

A row of a `yelt` table is a loss alone (YELT1, a flat vector, years numbered by position), `[year loss]` (YELT2), or `[year loss prob]` (YELT3). Zero-loss years must be listed, since they set the probability of no loss. Supplied probabilities must sum to one. Units, declared with `unit NAME` as in Using elt, are aligned by year and summed within each year, which keeps whatever dependence the simulation produced; every unit must supply the same complete set of years. An annual loss does not identify its occurrences, so a YELT takes aggregate reinsurance only.

The DecL clause `yelt`[1] builds an aggregate from a YELT. The reports `elt_description`, `elt_explanation`, and `elt_df`, described in Using elt, apply to a YELT too. File columns are matched by name as in Using elt with files: `year`, `trial` or `simulation_id` for the year, `loss` or `annual_loss` for the loss, and `prob` or `probability` for the probability.

```
agg Annual yelt [0 0 100 300]                        # YELT1: losses only
agg Paired yelt [[1 0] [2 0] [3 100] [4 300]]        # YELT2: [year loss]
agg Weighted yelt [[1 0 .5] [2 100 .3] [3 300 .2]] aggregate net of 150 xs 50
agg Sim yelt "sim.parquet"
```

```
import tempfile
from pathlib import Path
import numpy as np
import pandas as pd
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.43.1
```

A year loss table of four equally likely years

The table `yelt [0 0 100 300]` lists four years, each with probability 1/4, and gives the same aggregate as `dfreq [1] dsev [0 0 100 300]`. The probability of zero is one half, the mean is 100, and the variance is $(100^2 + 300^2)/4 - 100^2 = 15{,}000$, the population moment rather than the sample variance of 20,000.

[1] From aggregate 1.20.0

```
a = build('agg Annual yelt [0 0 100 300]')
qd(a)
b = build('agg Same dfreq [1] dsev [0 0 100 300]')
print(a.pmf(0), a.est_m, a.est_var, b.est_m, b.est_var)
```

```
Aggregate object: Annual. Frequency distribution empirical. Severity
dhistogram, [0, 300], bounded; atoms [0 100 300]. Updated with bucket
size 100 and log2 = 2. Validation: not unreasonable.

      Mean     SD     CV   Skew P01 Median P99
X
Freq     1      0      0
Sev    100 122.47 1.2247 0.8165   0      0 300
Agg    100 122.47 1.2247 0.8165   0      0 300
0.5 100.0 15000.0 100.0 15000.0
```

Supplied probabilities and an aggregate treaty

The aggregate `Weighted` is a YELT3 whose three years have probabilities 0.5, 0.3, and 0.2. Its `elt_df` lists each year of positive probability with `prob` and `loss`. The treaty `aggregate net of 150 xs 50` takes the outcomes 0, 100, and 300 to 0, 50, and 150, so the net mean is $0.3(50) + 0.2(150) = 45$ against a gross mean of 90.

```
w = build('agg Weighted yelt [[1 0 .5] [2 100 .3] [3 300 .2]] aggregate net of 150 xs 50')
wprint(w.elt_description)
display(GT(w.elt_df, float_format=',.4g'))
print(w.est_m)
```

```
loss table              YELT3, 3 years, 1 unit, supplied year probabilities
annual outcomes         3 outcomes, probability of zero 0.5, mean annual loss
                        90
grid                    bs 100, log2 2
checks                  11 pass
```

year	prob	loss
1	0.5	0
2	0.3	100
3	0.2	300

```
45.0
```

Units aligned by year

The aggregate `Years` has two units, A and B, and the table of unit B lists its years in a different order from unit A. The rows are matched by year and summed, so year 3 is $300 + 50 = 350$. The treaty `aggregate net of 150 xs 50` takes the totals 10, 100, and 350 to 10, 50, and 200, a net mean of 60.

```
y = build('''agg Years yelt unit A [[1 0 .5] [2 100 .3] [3 300 .2]]
                       yelt unit B [[3 50 .2] [1 10 .5] [2 0 .3]]
                       aggregate net of 150 xs 50''')
display(GT(y.elt_df, float_format=',.4g'))
print(y.est_m)
```

year	prob	loss
1	0.5	10
2	0.3	100
3	0.2	350

```
60.0
```

Two units from one CSV file

A long CSV file with a `Unit` column can hold several units. The file here holds the two units of the aggregate `Years`, under the headers `Trial`, `Annual Loss`, and `Probability`. The code writes it to a temporary directory and builds the aggregate there; the directory is removed when the `with` block ends, and the aggregate keeps what was read. The net mean is 60, the same as for the inline table. The provenance paragraph of `elt_explanation` lists the path, size, and SHA-256 digest of the file.

```
years = '''Trial,Unit,Annual Loss,Probability
1,A,0,.5
2,A,100,.3
3,A,300,.2
3,B,50,.2
1,B,10,.5
2,B,0,.3
'''
with tempfile.TemporaryDirectory() as d:
    path = Path(d, 'years.csv')
    path.write_text(years)
    f = build(f'agg YearsFile yelt "{path.as_posix()}" aggregate net of 150 xs 50')
print(f.est_m, y.est_m)
wprint(f.elt_explanation.split('\n\n')[1])
```

```
60.0 60.0
Source 1, unit A: file <tmp>/tmp89r4hn72/years.csv (102 bytes, modified
2026-10-30T12:37:20, SHA-256 ab4649389ba4), YELT3, 3 rows. Source 1, unit B:
file <tmp>/tmp89r4hn72/years.csv (102 bytes, modified 2026-10-30T12:37:20,
SHA-256 ab4649389ba4), YELT3, 3 rows.
```

Ten thousand simulated years from a Parquet file

The usual source of a YELT is the output of a simulation. Here a loss occurs in about 30% of 10,000 simulated years, lognormal when it occurs, and the years are saved to a Parquet file with `year` and `loss` columns. The aggregate is the empirical law of the years, so under `aggregate net of 1000 xs 500` its gross and net means equal the sample means of the gross and net losses, 47.87 and 39.69.

```
rng = np.random.default_rng(2026)
n = 10_000
sim = pd.DataFrame({'year': np.arange(1, n + 1),
                    'loss': np.round(rng.lognormal(4, 1.5, n) * (rng.random(n) < 0.3))})
with tempfile.TemporaryDirectory() as d:
    path = Path(d, 'sim.parquet')
    sim.to_parquet(path, index=False)
    s = build(f'agg Sim yelt "{path.as_posix()}" aggregate net of 1000 xs 500')
wprint(s.elt_description)
net = sim.loss - (sim.loss - 500).clip(0, 1000)
print(f'{sim.loss.mean():.4f} {net.mean():.4f} {s.est_m:.4f}')
```

```
loss table               YELT2, 10,000 years, 1 unit, equally likely years
annual outcomes          577 outcomes, probability of zero 0.6982, mean annual
                         loss 47.8732
grid                     bs 1, log2 14
checks                   11 pass
47.8732 39.6932 39.6932
```

Refused programs

Occurrence terms on a YELT are refused, because an annual loss does not identify its occurrences. Probabilities that do not sum to one, a unit missing a year, a year listed twice, and two units giving a year different probabilities are refused too. Each refusal is a `ValueError`, and the table checks raise its subclass `LossTableError`.

```
refuse('agg R1 yelt [0 0 100 300] occurrence net of 50 xs 50',
       'agg R2 yelt [[1 0 .5] [2 100 .3] [3 300 .1]]',
       'agg R3 yelt unit A [0 100 300] yelt unit B [10 20]',
       'agg R4 yelt [[1 0] [1 100] [2 300]]',
       'agg R5 yelt unit A [[1 0 .5] [2 100 .5]] yelt unit B [[1 0 .4] [2 100 .6]]')
```

```
DecL: a yelt holds annual outcomes, so occurrence reinsurance cannot apply: an
annual total does not identify its occurrences. Write annual terms as 'aggregate
net of ...', or declare the events with elt for occurrence terms.
R2: yelt source 1 (unit Total): the probabilities sum to 0.9, not one. Each row
is a year with its probability; counts, rates and weights are not probabilities.
R3: every unit supplies the same complete set of years, zero-loss years
included; yelt source 2 (unit B) has 2 years against 3 in yelt source 1 (unit A)
(missing years 3).
R4: yelt source 1 (unit Total): a year appears more than once; see row 1 (id 1),
row 2 (id 1).
R5: yelt source 2 (unit B) gives a year a different probability from yelt source
1 (unit A); a year has one probability, whatever its units.
```

Using `zm` and `zt`

Zero-truncated and zero-modified claim counts, and the two geometric distributions

Stephen J. Mildenhall

2026-11-01

A zero-modified count resets the probability of no claims and rescales the rest. If N has $p_k = \Pr(N = k)$, the modified count N^M has $\Pr(N^M = 0) = p_0^M$ and $\Pr(N^M = k) = c\, p_k$ for $k \geq 1$, with $c = (1 - p_0^M)/(1 - p_0)$, so $\mathsf{P}N^M = c\,\mathsf{P}N$. Zero truncation is the case $p_0^M = 0$. The modified counts form the (a, b, 1) class. They are used to fit a count whose probability of no claims differs from that of its family. In DecL the suffix `zt` after the frequency truncates the count, and `zm 0.5` modifies it to have $p_0^M = 1/2$. Both apply to the Poisson, binomial, negative binomial, geometric, and logarithmic counts.

The modification moves the mean, so the claim count in the exposure clause could set either the realized mean $\mathsf{P}N^M$ or the base mean $\mathsf{P}N$. The claim count sets the realized mean: `4 claims ... poisson zt` has a mean of four claims, and the base mean is solved for. A trailing `!` gives the textbook reading instead, where the clause sets the base mean and the realized mean is an output.

The geometric distribution has two conventions. The default `geometric` counts failures before the first success, on 0, 1, 2, ..., with $\Pr(G = k) = p(1 - p)^k$ and mean $(1 - p)/p$, so a mean of n gives $p = 1/(n + 1)$. The number of trials T, on 1, 2, ..., has $\Pr(T = k) = p(1 - p)^{k-1}$ and mean $1/p$. By memorylessness $G \mid G \geq 1$ has the law of $1 + G$, which is the law of T. The trials geometric is therefore the zero-truncated failures geometric, and it is written `geometric zt`.

```
agg PoZT 4 claims dsev [1] poisson zt          # mean 4, P(N = 0) = 0
agg PoZM 4 claims dsev [1] poisson zm 0.5      # mean 4, P(N = 0) = 1/2
agg PoBase 4 claims dsev [1] poisson zt !      # base mean 4, mean 4.0746
agg Trials 4 claims dsev [1] geometric zt      # trials, p = 1/4, mean 4
```

```
import warnings
import numpy as np
import pandas as pd
from scipy import stats
from aggregate import build, qd, __version__
from style import wprint, GT, refuse, figures
figures()
print(__version__)
```

```
1.60.0
```

With severity `dsev [1]` the aggregate is the count itself, so `density_df.p_total` is the probability mass function of N. A helper tabulates $\Pr(N = 0)$, $\Pr(N = 1)$, the realized mean `n`, the base mean, and the mean of the computed distribution. The grid is given explicitly, since the automatic one is too short for a geometric tail.

```
def counts(*clauses, n=4):
    rows = {}
    for clause in clauses:
        a = build(f'agg Count {n} claims dsev [1] {clause}', bs=1, log2=8)
        p = a.density_df.p_total
        rows[clause] = [p[0], p[1], a.n, a.base_mean, a.est_m]
    return pd.DataFrame.from_dict(rows, orient='index', columns=[
        'P(N=0)', 'P(N=1)', 'mean', 'base mean', 'computed mean']
        ).rename_axis('clause')
```

Zero-truncated and zero-modified Poisson counts

For a Poisson count with four claims in the exposure clause, a bare `zt` or `zm` keeps the mean at four and solves for the base mean. With `!` the base mean is four and the realized mean moves instead, up under truncation and down when the probability of no claims rises to 1/2. Keeping a mean of four claims with half the outcomes at zero needs a base mean near 8.

```
po = counts('poisson', 'poisson zt', 'poisson zt !', 'poisson zm 0.5',
            'poisson zm 0.5 !')
GT(po)
```

clause	P(N=0)	P(N=1)	mean	base mean	computed mean
poisson	0.0183	0.0733	4.0000	4.0000	4.0000
poisson zt	0.0000	0.0793	4.0000	3.9207	4.0000
poisson zt !	0.0000	0.0746	4.0746	4.0000	4.0746
poisson zm 0.5	0.5000	0.0013	4.0000	7.9973	4.0000
poisson zm 0.5 !	0.5000	0.0373	2.0373	4.0000	2.0373

The zero-truncated Poisson with base mean λ has mean $\lambda/(1-e^{-\lambda})$. The base mean 3.9207 solved for by `poisson zt` gives a mean of 4, and the base mean 4 of `poisson zt !` gives 4.0746, as in the table.

```
for lam in [po.loc['poisson zt', 'base mean'], 4]:
    print(f'{lam:.6f} {lam / (1 - np.exp(-lam)):.6f}')
```

```
3.920690 4.000000
4.000000 4.074629
```

Zero modification with a loss or premium exposure

The exposure clause of a zero-modified count can also be an expected loss or a premium, such as `400 loss`. A bare `zm` matches the stated loss: four claims of 100 give 400. With `!` the mean count moves away from the target, to 2.0373 claims and a loss of 203.73, and a `ZeroModifiedExposureWarning` gives the loss delivered.

```
with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter('always')
    for program in ['agg Loss 400 loss dsev [100] poisson zm 0.5',
                    'agg Loss 400 loss dsev [100] poisson zm 0.5 !']:
        a = build(program, bs=100, log2=8)
        print(f'{a.n:.4f} {a.est_m:.2f}')
for w in caught:
    wprint(f'{w.category.__name__}: {w.message}')
```

```
4.0000 400.00
2.0373 203.73
ZeroModifiedExposureWarning: Loss: the exposure clause states a monetary target
of 400, but zm/zt shifts the mean off it: E[N] 4 -> 2.03731 delivers 203.731.
Drop the ! from the zm/zt clause to pin the target instead.
```

Failures and trials geometric counts

The default `geometric`, counting failures, with mean 4 has $p = 1/5$, so $\Pr(N = 0) = 1/5$. The count `geometric zt` with mean 4 is the trials geometric with $p = 1/4$: $\Pr(N = 1) = 1/4$, and the failures geometric behind it has mean 3. With `!` the base mean is 4, $p = 1/5$, and the trials mean is 5. A zero-modified geometric is 0 with probability p_0^M and $1 + G$ otherwise, so its mean is $(1 - p_0^M)(1 + \text{base mean})$, here $(3/4)(1 + 13/3) = 4$.

```
geo = counts('geometric', 'geometric zt', 'geometric zt !', 'geometric zm 0.25')
GT(geo)
```

clause	P(N=0)	P(N=1)	mean	base mean	computed mean
geometric	0.2000	0.1600	4	4.0000	4.0000
geometric zt	0.0000	0.2500	4	3.0000	4.0000
geometric zt !	0.0000	0.2000	5	4.0000	5.0000
geometric zm 0.25	0.2500	0.1406	4	4.3333	4.0000

The distribution `scipy.stats.geom` is the trials form, and `geometric zt` matches it to within 10^{-17}. Shifting the failures geometric with mean 3 up by one gives the same law, as memorylessness requires.

```
k = np.arange(1, 50)
t = build('agg Trials 4 claims dsev [1] geometric zt',
          bs=1, log2=8).density_df.p_total
g = build('agg Failures 3 claims dsev [1] geometric',
          bs=1, log2=8).density_df.p_total
print(np.abs(t.loc[k].values - stats.geom(1/4).pmf(k)).max())
print(np.abs(t.loc[k].values - g.loc[k - 1].values).max())
```

```
7.15573433840433e-18
8.964996713739515e-18
```

Refused programs

A zero-truncated count cannot have a mean of one claim or fewer, since every outcome is at least one, and the error message gives the `!` form as an alternative. A count outside the (a, b, 1) class, such as `fixed`, has no zero modification. The value given to `zm` is a probability $\Pr(N = 0)$ that must lie in $[0, 1)$, since 1 leaves no probability for a claim.

```
refuse('agg X 0.5 claims dsev [1] poisson zt',
       'agg X 1 claim dsev [1] geometric zt',
       'agg X 4 claims dsev [1] fixed zt',
       'agg X 4 claims dsev [1] poisson zm 1',
       'agg X 4 claims dsev [1] poisson zm -0.2')
```

```
solve_base_mean: 'poisson' with p0 = 0 cannot reach a mean of 0.5; zero
modification admits only E[N] > 1. Append ! to read the count as the un-modified
base mean and let the modification shift it, or raise the claim count.
solve_base_mean: 'geometric' with p0 = 0 cannot reach a mean of 1; zero
modification admits only E[N] > 1. Append ! to read the count as the un-modified
base mean and let the modification shift it, or raise the claim count.
Zero modification not implemented for fixed
zm: P(N = 0) must lie in [0, 1), not 1.0
zm: P(N = 0) must lie in [0, 1), not -0.2
```