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.
```