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
```