# `proximal-lq`: Projected Gradient Descent for Simplex-Constrained Linear Least Squares

Thomas Schmelzer*

August 25, 2026

**Abstract**

`proximal-lq` solves $\min_{x\in\Delta} \frac{1}{2}\|Ax-b\|_2^2$, where $\Delta$ is the probability simplex, with projected gradient descent and the exact $O(n \log n)$ simplex projection of Duchi et al. (2008). The package is two functions and roughly two hundred lines of NumPy; this paper states what they compute, the step size and stopping rule they actually use, and how they behave on the problems in the test suite. Three things are worth recording. The implementation takes $\tau = 1/(2L)$ rather than the $\tau = 1/L$ its own development notes specify — both are convergent, but the shipped constant is half the classical one and costs roughly a factor of two in iterations. The parameter named `eps_rel` is compared against an absolute quantity, the norm of the last step, so tolerances do not scale with problem size. And the portfolio fixture the test suite validates against solves a least-squares fit of $\Sigma x \approx \mathbf{1}$, which we show is a different portfolio from the long-only minimum-variance one — variance 0.3087 against 0.0421 on the same covariance matrix — a distinction easy to lose when the two are described with the same words. The solver itself converges in tens of iterations and reproduces the reference solution to $4.4 \times 10^{-6}$ in $\ell_1$.

## 1 Introduction

Many estimation problems are least squares with the answer constrained to be a set of proportions: non-negative, summing to one. Long-only fully-invested portfolio weights (Markowitz, 1952) are the canonical example; mixture weights, non-negative sparse codes and discrete probability estimates are the same shape. Formally, with $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$,

$$\min_{x\in\mathbb{R}^n} \; f(x) = \tfrac{1}{2}\|Ax-b\|_2^2 \quad \text{subject to} \quad x \in \Delta, \qquad \Delta = \Big\{x \in \mathbb{R}^n : x \geq 0,\ \textstyle\sum_i x_i = 1\Big\}. \tag{1}$$

This is a convex quadratic program, so a general-purpose solver (Diamond and Boyd, 2016) will handle it. The reason not to use one is that Eq. (1) has structure a general solver cannot exploit: the constraint set is simple enough that projecting onto it is cheaper than one gradient evaluation. Projection onto $\Delta$ costs $O(n \log n)$ (Duchi et al., 2008), against $O(n^2)$ for a single gradient once the Gram matrix is formed. When the projection is the cheap step, projected gradient descent is the natural method: no factorisations, no interior point, no KKT system, and an iterate that is feasible after the first step and stays feasible.

`proximal-lq` is that method, kept deliberately small — two public functions, NumPy (Harris et al., 2020) as the only dependency. This paper documents it precisely enough to be used with confidence, which includes being explicit about the three places where the shipped behaviour differs from what a reader of the documentation would assume.

*`https://github.com/tschm/proximal`

## 2 Algorithm

### 2.1 Proximal gradient descent

Write Eq. (1) as the sum of a smooth part and an indicator,

$$\min_x \; f(x) + g(x), \qquad f(x) = \tfrac{1}{2}\|Ax - b\|_2^2, \qquad g(x) = \iota_\Delta(x) = \begin{cases} 0 & x \in \Delta \\ +\infty & \text{otherwise.} \end{cases} \tag{2}$$

The proximal gradient method — forward–backward splitting (Parikh and Boyd, 2014) — iterates

$$x^{k+1} = \operatorname{prox}_{\tau g}(x^k - \tau \nabla f(x^k)). \tag{3}$$

For an indicator function the proximal operator is the Euclidean projection, so Eq. (3) is exactly projected gradient descent, $x^{k+1} = P_\Delta(x^k - \tau \nabla f(x^k))$.

The gradient is

$$\nabla f(x) = A^\top (Ax - b) = Gx - c, \qquad G = A^\top A, \quad c = A^\top b, \tag{4}$$

and $G$ and $c$ are formed once, outside the loop. Each iteration then costs one matrix–vector product, $O(n^2)$, plus one projection, $O(n \log n)$ — the problem's $m$ enters only in the $O(mn^2)$ setup.

### 2.2 Step size

$\nabla f$ is Lipschitz with constant $L = \|A^\top A\|_2 = \sigma_{\max}(A)^2$, the spectral norm of the Gram matrix. Projected gradient descent converges for any fixed $\tau \in (0, 2/L)$, and the textbook choice $\tau = 1/L$ gives

$$f(x^k) - f(x^\star) \;\le\; \frac{\|x^0 - x^\star\|_2^2}{2\tau k} \tag{5}$$

— the $O(1/k)$ rate (Nesterov, 2004; Beck and Teboulle, 2009).

The implementation uses

$$\tau = \frac{1}{2L}, \tag{6}$$

half the classical value, falling back to $\tau = 1$ when $L < 10^{-15}$ so that an all-but-zero $A$ does not divide by zero. This is a discrepancy worth recording: the package's development notes state $\tau = 1/L$, while the code takes $0.5/L$. Both lie inside the convergent range, so nothing is broken, but by Eq. (5) the shipped constant roughly doubles the iteration count for a given accuracy — a conservative choice, not a wrong one, and the numbers in Section 3 are the numbers it produces.

### 2.3 Projection onto the simplex

The inner problem is

$$P_\Delta(v) = \operatorname*{arg\,min}_{x \ge 0,\, \sum_i x_i = s} \tfrac{1}{2}\|x - v\|_2^2. \tag{7}$$

Its solution is a soft threshold: $x_i = \max(v_i - \theta, 0)$ for the unique $\theta$ making the sum $s$. The KKT conditions give this immediately — the multiplier on the equality constraint is the same scalar $\theta$ for every coordinate, and the non-negativity multipliers are active exactly where $v_i \le \theta$. The only work is finding $\theta$, and sorting makes it explicit.

---

**Algorithm 1.** Simplex projection (Duchi et al., 2008)

---

**Input:** $v \in \mathbb{R}^n$, radius $s > 0$.

1: $\mu \leftarrow \mathrm{sort}(v)$, in decreasing order.
2: $\theta_j \leftarrow (\sum_{i \le j} \mu_i - s)/j$ for each $j$ — the running mean of the top $j$ entries, offset by $s$.
3: $\rho \leftarrow \max\{j : \mu_j > \theta_j\}$.
4: Return $x_i = \max(v_i - \theta_\rho, 0)$ for each $i$.

---

Algorithm 1 is exact — not an approximation or an inner iteration — and costs $O(n \log n)$, dominated by the sort, with $O(n)$ extra space. The implementation is a four-line vectorised transcription. Linear-time variants exist (Condat, 2016); at the sizes this package targets the sort is not the bottleneck.

Two properties inherited from exactness are worth naming, because the package's property tests assert them: the projection is idempotent, and it is the identity on points already in $\Delta$. A radius $s \neq 1$ is supported via the `rad` argument, which makes the same routine a projection onto a scaled simplex.

### 2.4 Stopping rule

The iteration halts when the step becomes small,

$$\|x^k - x^{k+1}\|_2 \;\le\; \varepsilon, \tag{8}$$

or when a cap of `max_iter` iterations is reached. For a projected gradient iteration this is a defensible surrogate for optimality: $x$ is optimal precisely when it is a fixed point of Eq. (3), so a vanishing step means a vanishing fixed-point residual.

Two caveats. First, the argument is named `eps_rel` and documented as a "relative stopping tolerance", but Eq. (8) compares an absolute norm: nothing divides by $\|x^k\|$, by $n$, or by the objective. A tolerance therefore does not transfer between problems of different scale, and on a large-$n$ problem the same $\varepsilon$ is a stricter test per coordinate. Second, hitting `max_iter` is silent — the last iterate is returned with no flag, warning, or residual, so a caller who needs to know whether Eq. (8) was met must re-measure it.

### 2.5 Initialisation

The primal variable starts from a standard normal draw, seeded via the `seed` argument. The draw is not in $\Delta$ — it is not even non-negative — which is harmless, because the first projection in Eq. (3) maps it into $\Delta$ and every subsequent iterate stays there. The problem is convex, so the optimum does not depend on the seed; the seed exists to make the iteration count and any tie-breaking reproducible.

## 3 Numerical results

### 3.1 Validation against a reference solution

The test suite pins the solver against the covariance matrix from the critical-line algorithm dataset of Bailey and López de Prado (2013) — ten assets, condition number 9.21 — solving Eq. (1) with $A = \Sigma$ and $b = \mathbf{1}$. The reference weight vector is reproduced to $\|x - x_{\mathrm{ref}}\|_1 = 4.4 \times 10^{-6}$ at the default tolerance, with seven of the ten weights driven to exactly zero.

Table 1 shows the iteration count growing roughly linearly in the number of digits requested — about 13 iterations per two decades — which is the geometric behaviour expected of a fixed-step projected gradient on a strongly convex problem, better than the $O(1/k)$ worst case of Eq. (5). The absolute counts are small: the whole solve is tens of matrix–vector products.

Table 1: Iterations to satisfy Eq. (8) on the ten-asset reference problem, at $\tau = 1/(2L)$.

| $\varepsilon$ | Iterations | Final step norm |
|---|---|---|
| $10^{-4}$ | 23 | $8.69 \times 10^{-5}$ |
| $10^{-6}$ | 36 | $8.14 \times 10^{-7}$ |
| $10^{-8}$ | 49 | $7.62 \times 10^{-9}$ |

Table 2: Random dense problems, $A \in \mathbb{R}^{2n \times n}$ with i.i.d. standard normal entries, default settings. *Non-zeros* counts weights above $10^{-9}$.

| $n$ | Wall time | $\sum_i x_i$ | Non-zeros |
|---|---|---|---|
| 10 | 1.1 ms | 1.0000000000 | 7 |
| 100 | 1.8 ms | 1.0000000000 | 29 |
| 1000 | 51.5 ms | 1.0000000000 | 91 |

### 3.2 Scaling and sparsity

Table 2 reports single-threaded timings on a laptop; they are indicative, not a benchmark. Two things in it matter more than the times.

The sum constraint holds to ten decimal places at every size, which is a consequence of the projection being exact rather than of the tolerance: every iterate is feasible, so feasibility does not degrade with the number of iterations the way it does for a penalty or an augmented-Lagrangian scheme.

The solutions are sparse, and increasingly so in relative terms: 91 non-zeros out of 1000 variables. This is intrinsic to the geometry, not a regulariser someone added. The unconstrained least-squares solution of a random over-determined system has no zeros; projecting onto the simplex thresholds every coordinate below $\theta$ to exactly zero, and the optimum of a linear objective over a polytope sits at a face. Users should expect exact zeros, and should not read them as evidence of a sparsity penalty.

### 3.3 What the portfolio fixture is not

The reference problem of Section 3 substitutes a covariance matrix for $A$ and $\mathbf{1}$ for $b$, so it minimises $\frac{1}{2}\|\Sigma x - \mathbf{1}\|_2^2$ over the simplex. It is tempting — and both the package's example script and its README invite the reading — to call the result an optimal portfolio without qualification. It is worth being precise about which portfolio, because it is not the one most readers will assume.

Table 3 puts the two side by side. The least-squares fixture yields a portfolio whose variance is more than seven times that of the minimum-variance portfolio on the same inputs, and the two weight vectors are far apart ($\ell_1$ distance 1.81 out of a maximum of 2). Solving $\Sigma x \approx \mathbf{1}$ in least squares recovers a minimum-variance-like direction only when the unconstrained solution $\Sigma^{-1}\mathbf{1}$ happens to be feasible for the simplex — that is, non-negative — and here it is not, at which point the least-squares residual and the variance are simply different objectives with different minimisers.

Nothing about the solver is at fault; this is a statement about the fixture. Note also that minimum variance is itself in reach of this package: with $A$ any matrix satisfying $A^\top A = \Sigma$ — a Cholesky or symmetric square-root factor — and $b = 0$, Eq. (1) *is* the long-only minimum-variance problem. The second row of Table 3 was produced by iterating this package's own projection on $\nabla f(x) = \Sigma x$.

Table 3: Two long-only fully-invested portfolios on the same ten-asset covariance matrix. Both are computed with this package's projection.

| | Objective solved | Variance $x^\top \Sigma x$ |
|---|---|---|
| Least-squares fixture | $\frac{1}{2}\|\Sigma x - \mathbf{1}\|_2^2$ | 0.30865 |
| Minimum variance | $\frac{1}{2} x^\top \Sigma x$ | 0.04212 |

# 4 Limitations

**No acceleration.** The iteration is plain proximal gradient. Nesterov momentum (Beck and Teboulle, 2009) would improve the worst-case rate from $O(1/k)$ to $O(1/k^2)$ at the cost of one extra vector of state, and the conservative step of Eq. (6) leaves further headroom: a backtracking line search would adapt $\tau$ to the local curvature instead of to a global bound. Neither is implemented.

**No convergence certificate.** The solver returns an array. It does not return the achieved residual, the iteration count, or whether Eq. (8) was ever satisfied, and `max_iter` is reached silently. For an automated pipeline this is the most consequential gap in the API: a non-converged answer is indistinguishable from a converged one.

**Dense and in-memory.** $G = A^\top A$ is formed explicitly, which is $O(n^2)$ storage and $O(mn^2)$ work regardless of any sparsity in $A$. For large sparse $A$ it would be cheaper to apply $A$ and $A^\top$ separately and never form $G$.

**Fixed problem shape.** Only Eq. (1) is supported: one matrix, one vector, one simplex. There is no weighted norm, no regularisation term, no box constraints beyond the simplex, no equality constraints other than the sum, and no warm start — each call begins from a fresh random point, so a sequence of related problems cannot reuse work.

# 5 Conclusion

Eq. (1) is a problem where the specialised method is both simpler and faster than the general one, because the projection is exact and cheap. Two hundred lines of NumPy solve the ten-asset reference problem to $10^{-8}$ in 49 iterations and a thousand-variable random problem in 51 ms, with feasibility holding to ten decimals at every iterate.

Three things a user should carry away, all of them documentation issues rather than defects. The step size is $1/(2L)$, half the value the development notes state, so iteration counts are about twice the textbook figure. The tolerance called `eps_rel` is absolute, so it does not transfer across problem scales. And the portfolio example solves a least-squares fit, not a minimum-variance problem — the same solver will do minimum variance, but only if it is handed a square-root factor of the covariance matrix and $b = 0$.

# A Reproducibility

All numbers were produced with `proximal-lq` 0.2.2 and the covariance matrix at `tests/proximal_lq/resources/CLA_Data.csv` (its first three rows are means and bounds and are skipped).

```
import numpy as np
from proximal_lq import prox_gradient, proj_simplex
```

```
S = np.genfromtxt("tests/proximal_lq/resources/CLA_Data.csv",
                  delimiter=",", skip_header=1)[3:]

x = prox_gradient(S, np.ones(10), seed=0)          # the fixture
print(x.round(5), x @ S @ x)

# minimum variance over the same simplex, via this package's projection
L = np.linalg.norm(S, 2)
y = np.ones(10) / 10
for _ in range(200_000):
    z = proj_simplex(y - (0.5 / L) * (S @ y))
    if np.linalg.norm(z - y) < 1e-14:
        y = z
        break
    y = z
print(y.round(5), y @ S @ y)
```

Iteration counts in Table 1 come from an instrumented copy of the library's loop — same Gram matrix, same step $0.5/L$, same projection — counting passes until Eq. (8) holds. Timings in Table 2 are single runs of `prox_gradient` on `rng.standard_normal((2*n, n))` with `seed=0`, measured with `time.perf_counter`.

## References


David H. Bailey and Marcos López de Prado. An open-source implementation of the critical-line algorithm for portfolio optimization. *Algorithms*, 6(1):169–196, 2013.

Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

Laurent Condat. Fast projection onto the simplex and the $\ell_1$ ball. *Mathematical Programming*, 158(1–2):575–585, 2016.

Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

John Duchi, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra. Efficient projections onto the $\ell_1$-ball for learning in high dimensions. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pages 272–279, 2008.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. Array programming with numpy. *Nature*, 585:357–362, 2020.

Harry Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77–91, 1952.

Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*. Springer, 2004.

Neal Parikh and Stephen Boyd. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):127–239, 2014.