Implementing the Critical Line Algorithm

Lorenzo Naranjo

March 2026

Introduction

The objective of this notebook is to compute the mean-variance frontier with short-sales constraints. Here I do so using the **Critical Line Algorithm** (CLA) of Markowitz (1959), although a simpler computational approach would be to solve the portfolio problem numerically on a fine grid of target returns.

The value of the CLA is that it makes the structure of the constrained frontier explicit. With no short sales, the frontier is not described by one global formula. Instead, it is built from a sequence of segments, each corresponding to a fixed active set of assets with positive weights. On each such segment, the portfolio weights and KKT multipliers are affine functions of the target return μ. The frontier changes only at **corner portfolios**, where the active set changes: an asset leaves when its weight falls to zero, and an excluded asset enters when its shadow price falls to zero.

The supporting theory is developed in the Short-Sales Constraints notebook. The purpose of the code here is to show that theory numerically and make concrete the asset pricing results that the companion notebook develops.

The constrained problem

We solve the standard mean-variance problem with non-negativity constraints:

$$\min_{\mathbf{w}} \ \frac{1}{2}\mathbf{w}'\mathbf{V}\mathbf{w} \quad \text{s.t.} \quad \mathbf{w}'\mathbf{e} = \mu,\ \mathbf{w}'\boldsymbol{\iota} = 1,\ w_i \geq 0 \ \ \forall\, i,$$

where $\mathbf{V}$ is the $n \times n$ positive-definite covariance matrix, $\mathbf{e}$ the vector of expected returns, and $\boldsymbol{\iota}$ a vector of ones. The Lagrangian introduces three types of multipliers:

$$\mathcal{L} = \frac{1}{2}\,\mathbf{w}'\mathbf{V}\mathbf{w} - \lambda\,(\mathbf{w}'\mathbf{e} - \mu) - \gamma\,(\mathbf{w}'\boldsymbol{\iota} - 1) - \boldsymbol{\nu}'\mathbf{w},$$

where λ enforces the return target, γ enforces the budget constraint, and $\nu_i \geq 0$ enforces non-negativity on each weight. Because $\mathbf{V}$ is positive definite, the KKT first-order conditions are necessary and sufficient:

$$\mathbf{V}\mathbf{w} = \lambda\,\mathbf{e} + \gamma\,\boldsymbol{\iota} + \boldsymbol{\nu}, \qquad \mathbf{w}'\boldsymbol{\iota} = 1, \qquad \mathbf{w}'\mathbf{e} = \mu,$$

together with the complementary-slackness conditions $\nu_i \geq 0$, $w_i \geq 0$, and $\nu_i\, w_i = 0$ for every asset i.

Active set, reduced system, and corners

Complementary slackness splits the assets into two groups. The **active set**

$$S = \{i : w_i > 0\}$$

contains the assets that are currently held. For these assets the non-negativity constraint is slack, so $\nu_i = 0$. The **inactive set**

$$S^c = \{i : w_i = 0\}$$

contains the excluded assets. For them, $\nu_j \geq 0$ is the shadow value of the no-short-sales constraint: if $\nu_j > 0$, forcing $w_j = 0$ is still optimal; if $\nu_j = 0$, asset j is exactly at the point where it may enter.

Once S is fixed, the constrained problem reduces to an ordinary Markowitz problem on the active assets only. Restricting the stationarity condition to $i \in S$ and using $\nu_i = 0$ gives

$$\mathbf{V}_S\,\mathbf{w}_S = \lambda\,\mathbf{e}_S + \gamma\,\boldsymbol{\iota}_S,$$

and the return and budget constraints become

$$\mathbf{w}_S'\mathbf{e}_S = \mu, \qquad \mathbf{w}_S'\boldsymbol{\iota}_S = 1.$$

Solving the first equation for the active weights gives

$$\mathbf{w}_S = \lambda\,\mathbf{V}_S^{-1}\mathbf{e}_S + \gamma\,\mathbf{V}_S^{-1}\boldsymbol{\iota}_S.$$

Substituting this expression into the return and budget constraints leaves a 2×2 linear system in the multipliers λ and γ. To write that system compactly, define

$$A_S = \boldsymbol{\iota}_S'\mathbf{V}_S^{-1}\mathbf{e}_S, \quad B_S = \mathbf{e}_S'\mathbf{V}_S^{-1}\mathbf{e}_S, \quad C_S = \boldsymbol{\iota}_S'\mathbf{V}_S^{-1}\boldsymbol{\iota}_S, \quad D_S = B_S C_S - A_S^2.$$

Then

$$\mu = \lambda B_S + \gamma A_S, \qquad 1 = \lambda A_S + \gamma C_S.$$

Solving for the multipliers yields

$$\lambda(\mu) = \frac{C\mu - A}{D}, \qquad \gamma(\mu) = \frac{B - A\mu}{D}, \qquad D > 0.$$

Substituting back into $\mathbf{w}_S$ shows that each active weight is affine in target return:

$$w_i(\mu) = a_{w,i} + b_{w,i}\,\mu, \quad i \in S.$$

Thus, once the active set is fixed, the whole segment is described by affine functions of μ.

For an inactive asset $j \in S^c$, the multiplier is

$$\nu_j(\mu) = (\mathbf{V}\mathbf{w})_j - \lambda(\mu)e_j - \gamma(\mu),$$

which is also affine in μ because $\mathbf{w}(\mu)$, $\lambda(\mu)$, and $\gamma(\mu)$ are affine. This is the key simplification behind the CLA: every candidate exit is the zero of an active-weight line, and every candidate entry is the zero of an inactive-multiplier line.

As μ varies, the active set stays fixed over an interval, so the efficient portfolios on that interval lie on a single frontier segment. A **corner portfolio** is the endpoint of such a segment, where

one of those affine entry or exit conditions hits zero and the active set changes. The constrained frontier is therefore piecewise, with one segment for each active set and corners joining adjacent segments.

Algorithm outline

The CLA then sweeps from the highest feasible return downward, one corner at a time:

1. **Seed**: start at the maximum-return corner (100 % in the highest-mean asset) and find a valid two-asset active set.
2. **Build segment**: solve the reduced KKT system for the current S to obtain the affine weight and multiplier formulas.
3. **Scan for events**: compute all candidate entry and exit returns below the current return.
4. **Select next corner**: the largest such crossing is the next corner μ^*.
5. **Update** S: remove assets whose weights hit zero and add assets whose shadow prices hit zero.
6. **Repeat** until no crossings remain or S becomes empty.

The implementation below translates each of these steps into a short, self-contained code cell.

Core Code

The code follows the derivation above: build the affine formulas on a fixed active set, compute entry and exit returns, and update the active set from corner to corner.

Imports and numerical tolerances

These imports and tolerances are used throughout. The tolerances control degeneracy checks, affine roots, progress from one corner to the next, and small feasibility errors.

```
import numpy as np
import matplotlib.pyplot as plt
import yfinance as yf
import pandas as pd
from typing import NamedTuple

EPS_DET = 1e-14
EPS_SLOPE = 1e-14
EPS_EVENT = 1e-10
EPS_ACTIVE = 1e-8
```

Data structure for one CLA segment

For a fixed active set, weights and KKT multipliers are affine in μ. `Segment` stores those coefficients for one frontier segment.

```
class Segment(NamedTuple):
    aw: np.ndarray
    bw: np.ndarray
    lam_a: float
    lam_b: float
    gam_a: float
    gam_b: float
```

Basic helpers

These are utility functions for evaluating segment weights, computing volatility, and loading annualized inputs.

```
def weights_at_mu(seg, mu):
    return seg.aw + seg.bw * mu

def sigma_of_weights(w, V):
    return np.sqrt(max(w @ V @ w, 0.0))

def load_annualized_inputs(tickers, start_date):
    prices = yf.download(
        tickers,
        start=start_date,
        interval='1mo',
        auto_adjust=True,
        progress=False,
    )['Close']
    ret = prices.pct_change().dropna()
    e = 12 * ret.mean().to_numpy()
    V = 12 * ret.cov().to_numpy()
    return e, V
```

Frontier scalars on a fixed active set

Given an active set S, the Markowitz scalars

$$A = \boldsymbol{\iota}_S' \mathbf{V}_S^{-1} \mathbf{e}_S,\ B = \mathbf{e}_S' \mathbf{V}_S^{-1} \mathbf{e}_S,\ C = \boldsymbol{\iota}_S' \mathbf{V}_S^{-1} \boldsymbol{\iota}_S$$

summarize the reduced problem. This helper computes them from a single linear solve.

```
def frontier_scalars(Vs, es):
    ones = np.ones(len(es))
    X = np.linalg.solve(Vs, np.column_stack([es, ones]))
    x_e, x_1 = X[:, 0], X[:, 1]
    A = ones @ x_e
```

```
    B = es @ x_e
    C = ones @ x_1
    return A, B, C
```

Build one segment

This is the fixed-active-set solve. It computes the affine formulas for $\lambda(\mu)$, $\gamma(\mu)$, and $w(\mu)$ on the current segment.

```
def build_segment(active, e, V, n):
    Vs = V[np.ix_(active, active)]
    es = e[active]
    ones = np.ones(len(active))

    X = np.linalg.solve(Vs, np.column_stack([es, ones]))
    x_e, x_1 = X[:, 0], X[:, 1]
    A = ones @ x_e
    B = es @ x_e
    C = ones @ x_1
    D = B * C - A**2
    if D < EPS_DET:
        return None

    lam_a, lam_b = -A / D, C / D
    gam_a, gam_b = B / D, -A / D

    aw = np.zeros(n)
    bw = np.zeros(n)
    aw[active] = lam_a * x_e + gam_a * x_1
    bw[active] = lam_b * x_e + gam_b * x_1
    return Segment(aw, bw, lam_a, lam_b, gam_a, gam_b)
```

Event equations: when does an asset hit the boundary?

Each candidate corner comes from the zero of an affine function. Active assets exit when $w_i(\mu) = 0$; inactive assets enter when $\nu_j(\mu) = 0$.

```
def _affine_root(a, b):
    """Return -a/b if |b| is large enough, else None."""
    if abs(b) < EPS_SLOPE:
        return None
    return -a / b

def w_zero(i, seg):
    return _affine_root(seg.aw[i], seg.bw[i])

def nu_zero(j, seg, e, V):
    av = V[j] @ seg.aw - seg.lam_a * e[j] - seg.gam_a
    bv = V[j] @ seg.bw - seg.lam_b * e[j] - seg.gam_b
    return _affine_root(av, bv)
```

Seed segment at the top corner

The algorithm starts at the maximum-return corner. This routine looks for a valid two-asset seed segment consistent with that corner and the dual feasibility conditions.

```
def find_valid_seed(max_return_idx, e, V, n):
    top_mu = e[max_return_idx]
    for candidate in range(n):
        if candidate == max_return_idx:
            continue

        active = sorted([max_return_idx, candidate])
        seg = build_segment(active, e, V, n)
```

```
        if seg is None:
            continue

        w_top = weights_at_mu(seg, top_mu)
        if (
            abs(w_top[candidate]) > EPS_ACTIVE
            or abs(w_top[max_return_idx] - 1.0) > EPS_ACTIVE
        ):
            continue

        lam_top = seg.lam_a + seg.lam_b * top_mu
        gam_top = seg.gam_a + seg.gam_b * top_mu
        if all(
            V[idx] @ w_top - lam_top * e[idx] - gam_top >= -EPS_ACTIVE
            for idx in range(n)
            if idx not in active
        ):
            return active, seg

    return None, None
```

Collect entry/exit candidates from current segment

Given a segment, this function computes all admissible exit and entry returns below the current corner.

```
def find_events(active, n, seg, mu_current, e, V):
    active_set = set(active)
    events = [
        (mu_event, idx, 'exit')
        for idx in active
        if (mu_event := w_zero(idx, seg)) is not None
```

```
        and mu_event < mu_current - EPS_EVENT
    ]
    events += [
        (mu_event, idx, 'enter')
        for idx in range(n)
        if idx not in active_set
        and (mu_event := nu_zero(idx, seg, e, V)) is not None
        and mu_event < mu_current - EPS_EVENT
    ]
    return events
```

Main CLA loop

This is the main CLA loop. It builds a segment, finds the next corner, updates the active set, and repeats until no further corner is found.

```
def run_cla(e, V):
    n = len(e)
    max_return_idx = int(np.argmax(e))

    corners = [(e[max_return_idx], np.eye(n)[max_return_idx])]
    segments = []

    active, seed_seg = find_valid_seed(max_return_idx, e, V, n)
    if active is None:
        active = list(range(n))

    mu_current = e[max_return_idx]
    seg = seed_seg

    for _ in range(2 * n + 4):
        if seg is None:
```

```
            seg = build_segment(active, e, V, n)
        if seg is None:
            break

        candidates = find_events(active, n, seg, mu_current, e, V)
        if not candidates:
            break

        mu_next = max(candidates, key=lambda x: x[0])[0]
        events_at_corner = [evt for evt in candidates if abs(evt[0] - mu_next) <= EPS_
        segments.append((seg, mu_current, mu_next))

        w_corner = np.clip(weights_at_mu(seg, mu_next), 0, None)
        weight_sum = w_corner.sum()
        if weight_sum <= EPS_ACTIVE:
            break
        w_corner /= weight_sum
        corners.append((mu_next, w_corner))

        exits = {asset for _, asset, evt in events_at_corner if evt == 'exit'}
        enters = {asset for _, asset, evt in events_at_corner if evt == 'enter'}
        active = sorted((set(active) - exits) | enters)

        mu_current = mu_next
        seg = None
        if not active:
            break

    return corners, segments
```

Sample points on the constrained and unconstrained frontiers

The CLA returns corners and affine segment formulas. These helpers convert them into plot-ready points and compute the unconstrained benchmark frontier.

```
def evaluate_constrained_frontier(segments, V, grid_points):
    mu_con, sig_con = [], []
    for seg, mu_high, mu_low in segments:
        for mu_val in np.linspace(mu_high, mu_low, grid_points):
            w = weights_at_mu(seg, mu_val)
            if np.all(w >= -EPS_ACTIVE):
                mu_con.append(mu_val)
                sig_con.append(sigma_of_weights(w, V))
    return np.array(mu_con), np.array(sig_con)

def unconstrained_frontier(e, V):
    A, B, C = frontier_scalars(V, e)
    D = B * C - A**2
    if D < EPS_DET:
        raise ValueError('Degenerate unconstrained frontier: D is too small.')
    mu_unc = np.linspace(A / C - 0.08, e.max(), 600)
    var_unc = (B - 2 * A * mu_unc + C * mu_unc**2) / D
    sig_unc = np.sqrt(np.maximum(var_unc, 0.0))
    return mu_unc, sig_unc
```

Plotting helpers

These functions plot the constrained and unconstrained frontiers and optionally label the corner portfolios.

```
def corner_weight_label(w, tickers):
    parts = [f"{tk}:{wi:.2f}" for tk, wi in zip(tickers, w) if wi > 1e-3]
```

```
    return '\n'.join(parts)

def plot_frontiers(e, V, corners, segments, tickers, title, label_corner_weights=False
    mu_con, sig_con = evaluate_constrained_frontier(segments, V, grid_points=grid_poin
    mu_unc, sig_unc = unconstrained_frontier(e, V)

    fig, ax = plt.subplots(figsize=(7, 5))
    ax.plot(sig_unc * 100, mu_unc * 100, color='steelblue', lw=2, ls='--', label='Unco
    ax.plot(sig_con * 100, mu_con * 100, color='darkorange', lw=2, label=r'Constrained

    n_corners = max(len(corners), 1)
    for idx, (mu_c, w_c) in enumerate(corners):
        sc = sigma_of_weights(w_c, V)
        ax.scatter([sc * 100], [mu_c * 100], color='darkorange', zorder=5, s=48)
        if label_corner_weights:
            angle = 2 * np.pi * idx / n_corners
            radius = 16
            dx = int(np.round(radius * np.cos(angle)))
            dy = int(np.round(radius * np.sin(angle)))
            ha = 'left' if dx >= 0 else 'right'
            va = 'bottom' if dy >= 0 else 'top'
            ax.annotate(
                corner_weight_label(w_c, tickers),
                (sc * 100, mu_c * 100),
                textcoords='offset points',
                xytext=(dx, dy),
                ha=ha,
                va=va,
                fontsize=8,
                bbox=dict(boxstyle='round,pad=0.2', fc='white', ec='0.8', alpha=0.95),
                arrowprops=dict(arrowstyle='-', color='0.55', lw=0.8, shrinkA=4, shrin
            )
```

```
    ax.set_title(title)
    ax.set_xlabel(r'$\sigma$ (%)')
    ax.set_ylabel(r'$\mu$ (%)')
    ax.set_xlim(left=0)
    ax.legend(frameon=False, fontsize=9)
    ax.spines[['top', 'right']].set_visible(False)
    plt.tight_layout()
    plt.show()
```

Tabular summary of corner portfolios

This helper builds a table with return, risk, and weights for each corner portfolio.

```
def build_corner_weights_table(corners, tickers, V):
    rows = []
    for idx, (mu_c, w_c) in enumerate(corners):
        rows.append(
            {
                'Corner': f'C{idx + 1}',
                '  (%)': f'{mu_c * 100:.2f}',
                '  (%)': f'{sigma_of_weights(w_c, V) * 100:.2f}',
                **{tk: f'{w_c[i]:.4f}' for i, tk in enumerate(tickers)},
            }
        )
    return pd.DataFrame(rows).set_index('Corner')
```

Examples

These examples apply the same workflow to two asset universes. The four-asset case makes the corner portfolios easy to inspect; the ten-asset case shows the same algorithm on a larger uni-

verse. In each figure, compare the piecewise constrained frontier to the smooth unconstrained benchmark and relate each bend to a change in the active set.

An Example with Four Assets

With four assets, the corner labels remain readable. This makes it easy to see how the active set changes from one corner to the next.

```
tickers_4 = ['AAPL', 'BA', 'C', 'WMT']
start_date = '2000-01-01'
grid_points = 300

e_4, V_4 = load_annualized_inputs(tickers_4, start_date=start_date)
corners_4, segments_4 = run_cla(e_4, V_4)

plot_frontiers(
    e_4,
    V_4,
    corners_4,
    segments_4,
    tickers_4,
    title='CLA Frontier: AAPL, BA, C, WMT',
    label_corner_weights=True,
    grid_points=grid_points,
)
```

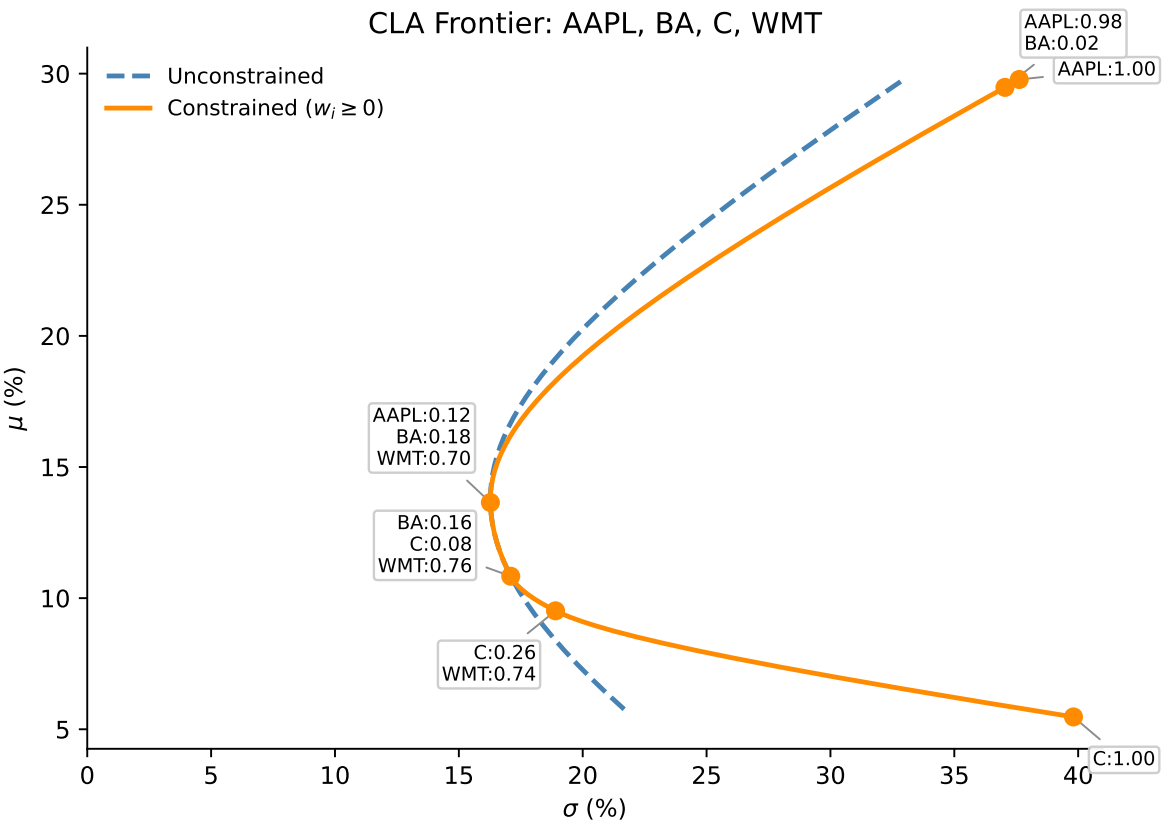

Figure 1: Constrained and unconstrained frontiers for AAPL, BA, C, and WMT. Corner dots are annotated with corner portfolio weights.

The table below lists the corner weights explicitly. Reading adjacent rows shows how portfolio composition changes from one corner to the next.

```
build_corner_weights_table(corners_4, tickers_4, V_4)
```

Table 1: Corner portfolios for AAPL, BA, C, and WMT.

Corner	μ (%)	σ (%)	AAPL	BA	C	WMT
C1	29.78	37.62	1.0000	0.0000	0.0000	0.0000
C2	29.48	37.05	0.9819	0.0181	0.0000	0.0000

Table 1: Corner portfolios for AAPL, BA, C, and WMT.

Corner	μ (%)	σ (%)	AAPL	BA	C	WMT
C3	13.65	16.27	0.1234	0.1804	0.0000	0.6962
C4	10.84	17.09	0.0000	0.1583	0.0797	0.7620
C5	9.52	18.90	0.0000	0.0000	0.2565	0.7435
C6	5.47	39.81	0.0000	0.0000	1.0000	0.0000

A More Realistic Universe

This example uses the same code on a ten-asset universe. Labels are omitted to keep the figure readable, but the logic is unchanged: each kink corresponds to a change in the active set.

```
tickers_10 = ['AAPL', 'BA', 'C', 'GE', 'GM', 'JNJ', 'KO', 'MAR', 'MMM', 'WMT']
e_10, V_10 = load_annualized_inputs(tickers_10, start_date=start_date)
corners_10, segments_10 = run_cla(e_10, V_10)

plot_frontiers(
    e_10,
    V_10,
    corners_10,
    segments_10,
    tickers_10,
    title='CLA Frontier: 10 Assets',
    label_corner_weights=False,
    grid_points=grid_points,
)
```

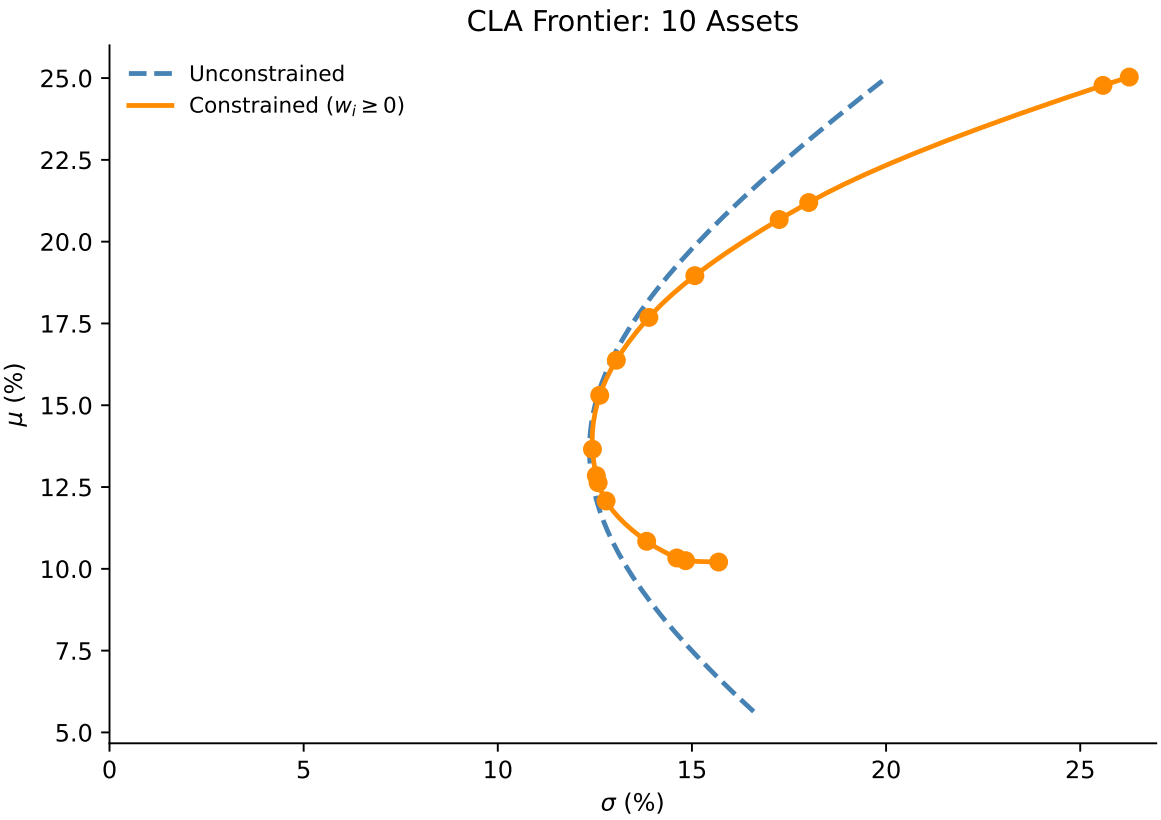

Figure 2: Constrained and unconstrained frontiers for 10 assets. Corner portfolios are shown as dots only (no labels).

CLA vs. Generic Solver

The last example compares the CLA with a generic quadratic-program solver. The CVXPY solution traces the frontier point by point, while the CLA builds it segment by segment from the entry and exit conditions. The two curves should coincide.

```
import warnings
import cvxpy as cp

n = len(e_10)
w_var = cp.Variable(n)
```

```
mu_target = cp.Parameter()
prob = cp.Problem(
    cp.Minimize(cp.quad_form(w_var, V_10)),
    [w_var @ e_10 == mu_target, cp.sum(w_var) == 1, w_var >= 0],
)

mu_grid = np.linspace(e_10.min(), e_10.max(), 300)
mu_cvx, sig_cvx = [], []
for mu_val in mu_grid:
    mu_target.value = mu_val
    with warnings.catch_warnings():
        warnings.simplefilter('ignore', UserWarning)
        prob.solve()
    if prob.status == 'optimal':
        mu_cvx.append(mu_val)
        sig_cvx.append(np.sqrt(w_var.value @ V_10 @ w_var.value))

mu_cla, sig_cla = evaluate_constrained_frontier(segments_10, V_10, grid_points=grid_po

fig, ax = plt.subplots(figsize=(7, 5))
ax.plot(sig_cla * 100, mu_cla * 100, color='darkorange', lw=2, label='CLA')
ax.scatter(
    np.array(sig_cvx) * 100,
    np.array(mu_cvx) * 100,
    color='steelblue', s=10, zorder=5, label='CVXPY',
)
ax.set_title('CLA vs. CVXPY: 10 Assets')
ax.set_xlabel(r'$\sigma$ (%)')
ax.set_ylabel(r'$\mu$ (%)')
ax.set_xlim(left=0)
ax.legend(frameon=False, fontsize=9)
ax.spines[['top', 'right']].set_visible(False)
```

```
plt.tight_layout()
plt.show()
```

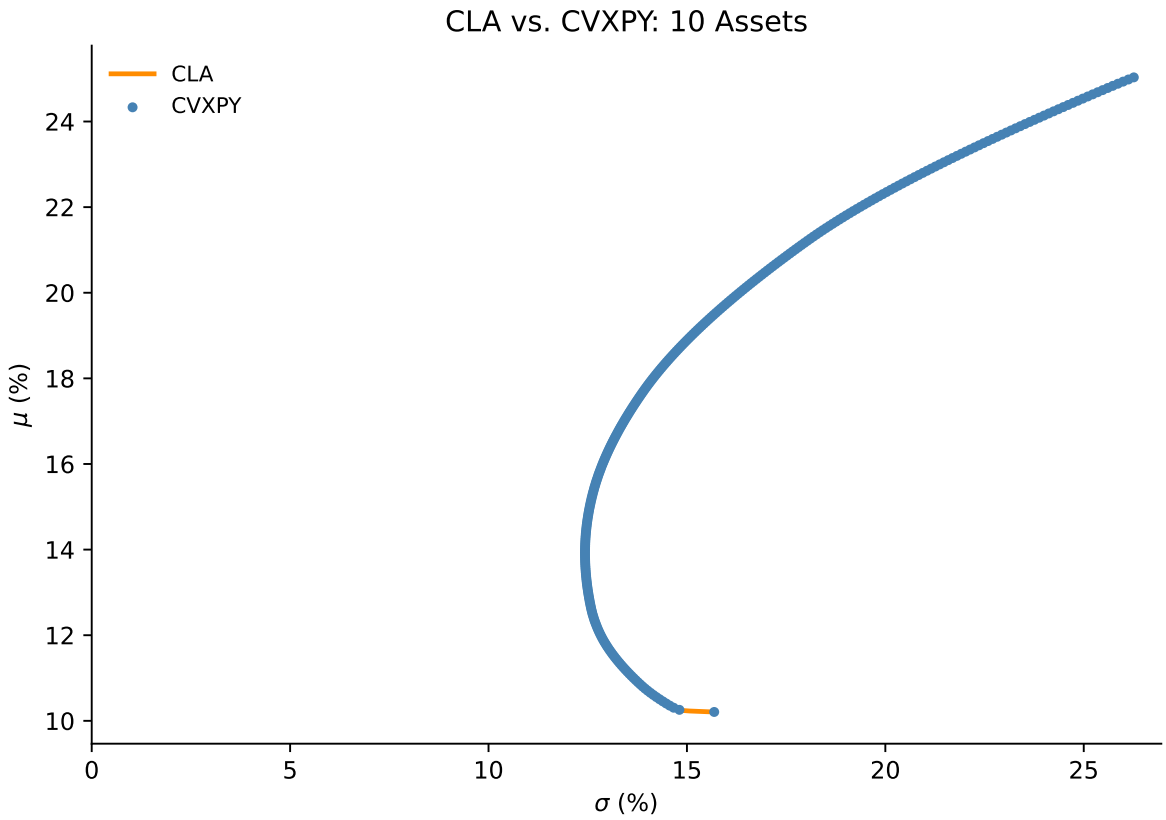

Figure 3: CLA frontier (solid) vs. CVXPY point-by-point solver (dots) for the 10-asset universe.

References

Markowitz, Harry M. 1959. *Portfolio Selection: Efficient Diversification of Investments*. Cowles Foundation Monograph 16. John Wiley & Sons.