

# Portfolio Optimization Using Linear Algebra

Quang Nguyen, Fall 2025

# What is Portfolio Optimization?

- Portfolio optimization is the process of finding the optimal asset allocation (meaning how much should be invested into each different asset) so that we can minimize the risk (measured as volatility) for a given level of target return
- Said differently, if I want to earn, say 10% a year on average on my investments, how much of my portfolio should I invest into stocks, bonds, gold, real estate, etc. so that I can earn 10% while minimizing the risk that the portfolio experiences
- The project is a practical implementation and extension of Harry Markowitz's Modern Portfolio Theory

# Linear Algebra Implementation

- Nearly every core step in the portfolio optimization pipeline involves matrix and vector operations
- Asset Returns represented as a vector
- Covariance of Assets (how assets move with one another) is represented in a matrix
- We want to solve for the following equation.

$$\min_w w^\top \Sigma w \quad \text{s.t.} \quad \mu^\top w = r^*, \quad 1^\top w = 1$$

- Our problem is a backward problem and our solution is a column vector of asset weights.

Ticker	
AGG	0.019116
BTC-USD	0.566391
DBC	0.060831
EEM	0.066203
EFA	0.071647
GLD	0.130344
IJH	0.093500
IWM	0.082059
SPY	0.132616
VNQ	0.051232

dtype: float64

# Asset Classes

---

```
# List of ETFs that are to represent their respective asset classes
```

```
tickers = ["SPY", # U.S. Large-Cap Equities  
          "IJH", # U.S. Mid Cap  
          "IWM", # U.S. Small Cap  
          "EFA", # International Developed Markets  
          "EEM", # Emerging Markets  
          "AGG", # U.S. Aggregate Bonds  
          "VNQ", # Real Estate  
          "DBC", # Commodities  
          "GLD", # Gold  
          "BTC-USD" # Bitcoin  
          ]
```



# Data Source

- 10 Years of historical data from the yfinance Python library

---

```
# Establish start and end date for pulling price data

end_date = datetime.today()
start_date = end_date - timedelta(days=10 * 365)
```

---

---

```
# Pull Adjusted Close data for the defined list of tickers over the specified time period

adj_close_df = yf.download(tickers=tickers,
                           start=start_date,
                           end=end_date,
                           auto_adjust=False
                           )["Adj Close"]
```

---

# Solving

- We can solve the entire optimization problem with a single mathematical formula, a closed-form matrix solution:

$$w(r^*) = \frac{\mu^\top \Sigma^{-1} \mu - r^* \mathbf{1}^\top \Sigma^{-1} \mu}{(\mathbf{1}^\top \Sigma^{-1} \mathbf{1})(\mu^\top \Sigma^{-1} \mu) - (\mathbf{1}^\top \Sigma^{-1} \mu)^2} \Sigma^{-1} \mathbf{1} + \frac{r^* \mathbf{1}^\top \Sigma^{-1} \mathbf{1} - \mathbf{1}^\top \Sigma^{-1} \mu}{(\mathbf{1}^\top \Sigma^{-1} \mathbf{1})(\mu^\top \Sigma^{-1} \mu) - (\mathbf{1}^\top \Sigma^{-1} \mu)^2} \Sigma^{-1} \mu$$

$$w(r^*) = \frac{C - r^* B}{D} \Sigma^{-1} \mathbf{1} + \frac{r^* A - B}{D} \Sigma^{-1} \mu$$

$$\begin{aligned}A &= \mathbf{1}^\top \Sigma^{-1} \mathbf{1} \\B &= \mathbf{1}^\top \Sigma^{-1} \mu \\C &= \mu^\top \Sigma^{-1} \mu \\D &= AC - B^2\end{aligned}$$

# Code Implementation

- Our target returns are stored in an array and ranged from 2% to 20%

```
mu = annualized_mean_return.values          # Vector of expected annualized returns
sigma = cov_annual.values                   # Covariance matrix annualized
target_returns = np.linspace(0.02, 0.20, 10) # Target returns (10 evenly spaced returns from 2% to 20%)

ones = np.ones(len(mu))                     # Ones will be used as a constraint for our optimization problem.
| | | | | | | | | | | | | | | | | | | | | # The sum the weights for all our assets should sum to 1.

inverse_sigma = np.linalg.inv(sigma)        # The inverse matrix or precision matrix of our Covariance matrix
| | | | | | | | | | | | | | | | | | | | | # tells us how each asset contributes to risk-adjusted combinations.
```

# Defining Our Scalars and Closed-form Matrix Solution Function

```
# Matrix multiplication to solve for Scalar Constants

A = ones @ inverse_sigma @ ones
B = ones @ inverse_sigma @ mu
C = mu @ inverse_sigma @ mu
D = A * C - B**2
```

Python

```
# Defining Closed-form Matrix Solution solution

def min_var_weights(r_star):
    w_minvar_target = ((C - r_star * B) / D) * (inverse_sigma @ ones) + ((r_star * A - B) / D) * (inverse_sigma @ mu)
    return w_minvar_target
```

Python

We iterate through the target returns as each target return will have its own respective asset weights

```
# Lists to store weights, portfolio returns, and portfolio volatility

weights_list = []
portfolio_returns = []
portfolio_vols = []
```

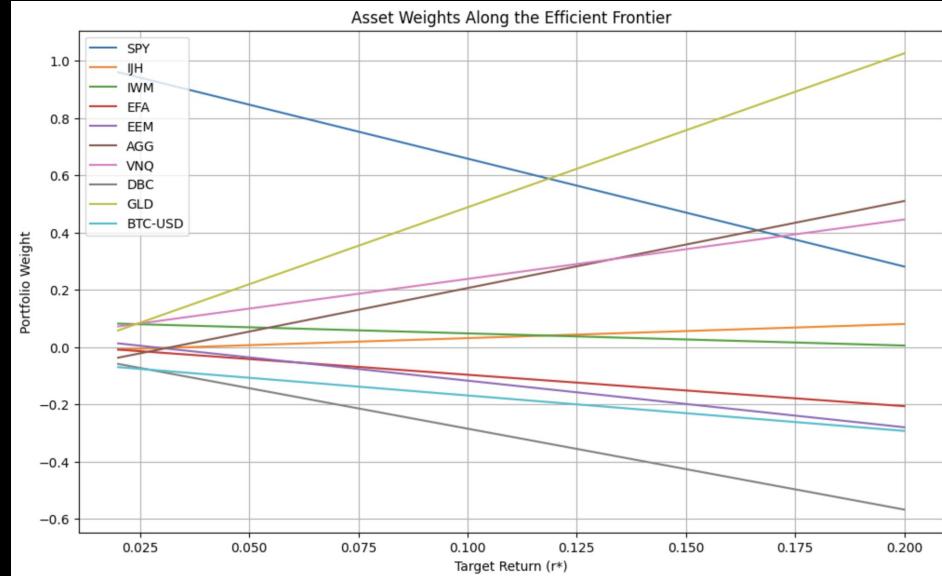
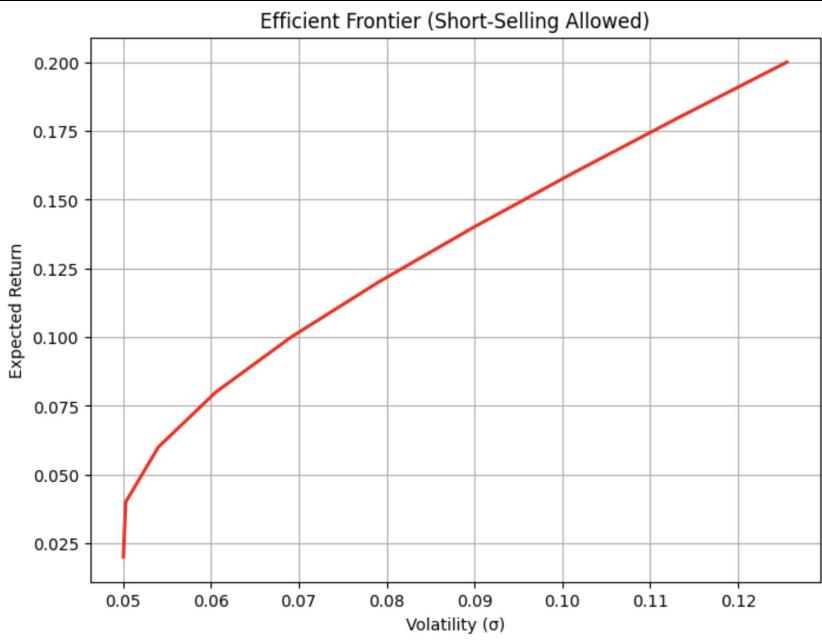
Python

```
# Iterate through the target returns list, for each target return,
# append respective weights and volatility to a list

for r_star in target_returns:
    w = min_var_weights(r_star)
    weights_list.append(w)
    port_return = mu @ w
    port_var = w.T @ sigma @ w
    portfolio_returns.append(port_return)
    portfolio_vols.append(np.sqrt(port_var))
```

Python

# Results (Unconstrained - Shorting Allowed)



# Results Continued. . .

	<b>2.0%</b>	<b>4.0%</b>	<b>6.0%</b>	<b>8.0%</b>	<b>10.0%</b>	<b>12.0%</b>	<b>14.0%</b>	<b>16.0%</b>	<b>18.0%</b>	<b>20.0%</b>
SPY	0.959868	0.884452	0.809036	0.733620	0.658203	0.582787	0.507371	0.431955	0.356538	0.281122
IJH	-0.008309	0.001546	0.011401	0.021255	0.031110	0.040964	0.050819	0.060674	0.070528	0.080383
IWM	0.081705	0.073198	0.064691	0.056184	0.047677	0.039170	0.030663	0.022156	0.013649	0.005142
EFA	-0.009043	-0.031000	-0.052957	-0.074913	-0.096870	-0.118827	-0.140784	-0.162740	-0.184697	-0.206654
EEM	0.012736	-0.019826	-0.052388	-0.084950	-0.117512	-0.150074	-0.182636	-0.215198	-0.247760	-0.280322
AGG	-0.037368	0.023446	0.084260	0.145074	0.205887	0.266701	0.327515	0.388329	0.449143	0.509956
VNQ	0.071895	0.113396	0.154898	0.196399	0.237900	0.279401	0.320903	0.362404	0.403905	0.445407
DBC	-0.058957	-0.115506	-0.172055	-0.228604	-0.285153	-0.341702	-0.398251	-0.454800	-0.511349	-0.567898
GLD	0.057769	0.165338	0.272906	0.380475	0.488043	0.595612	0.703180	0.810749	0.918317	1.025886
BTC-USD	-0.070298	-0.095045	-0.119792	-0.144539	-0.169286	-0.194033	-0.218780	-0.243527	-0.268275	-0.293022

Notice, we are shorting bitcoin at an increasing proportion, which seems counterintuitive because recall our return vector (right).

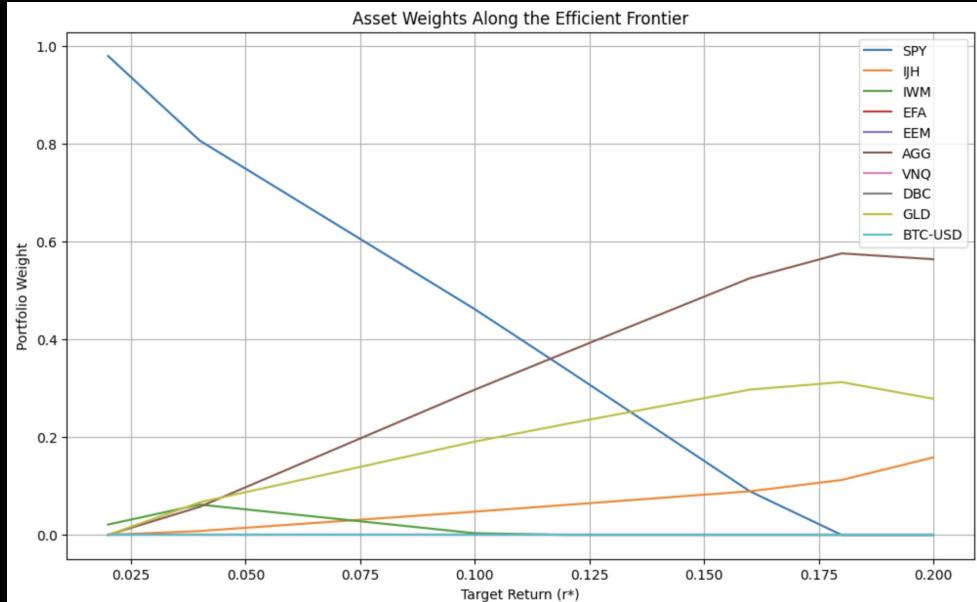
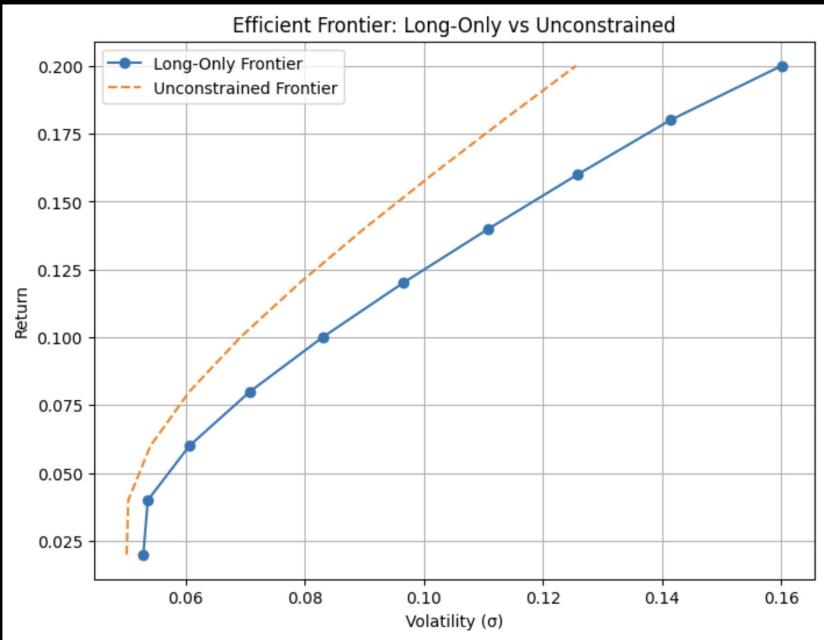
**Interpretation:** Although Bitcoin has extremely high returns, it also contributes very large variance and strong positive covariance with other risky assets.

Ticker	
AGG	0.019116
BTC-USD	0.566391
DBC	0.060831
EEM	0.066203
EFA	0.071647
GLD	0.130344
IJH	0.093500
IWM	0.082059
SPY	0.132616
VNQ	0.051232

# Many investment funds have a long-only mandate. . .

- Meaning no shorting is allowed
- As such, we also constructed portfolios that are long-only
- We used a solver, numerical convex optimization (CVXPY), to find the optimal weights

	2.0%	4.0%	6.0%	8.0%	10.0%	12.0%	14.0%	16.0%	18.0%	20.0%
SPY	0.978806	0.805874	0.691003	0.576132	0.461262	0.338366	0.213735	0.089104	0.000000	-0.000000
IJH	0.000000	0.007920	0.021157	0.034394	0.047631	0.061407	0.075300	0.089192	0.112250	0.158295
IWM	0.021194	0.062047	0.042521	0.022996	0.003470	0.000000	0.000000	0.000000	0.000000	0.000000
EFA	0.000000	-0.000000	-0.000000	-0.000000	-0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
EEM	0.000000	-0.000000	-0.000000	-0.000000	-0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
AGG	0.000000	0.057482	0.137287	0.217092	0.296897	0.373313	0.448997	0.524680	0.575389	0.563459
VNQ	0.000000	-0.000000	-0.000000	-0.000000	-0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
DBC	0.000000	-0.000000	-0.000000	-0.000000	-0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
GLD	0.000000	0.066678	0.108032	0.149387	0.190741	0.226915	0.261969	0.297023	0.312361	0.278246
BTC-USD	0.000000	-0.000000	-0.000000	-0.000000	-0.000000	0.000000	0.000000	0.000000	0.000000	0.000000



# Observations

- When we introduce the long-only constraint, for each target return, the portfolio needs to take on higher volatility to achieve the same target return.
  - This is because when unconstrained, short-selling allows for the hedging of correlated risk. Volatility-heavy assets (like Bitcoin) can be offsetted & covariance contribution cannot be canceled out.
- The long-only portfolio also makes use of fewer assets.
  - Unconstrained (20% Target Return) - Makes Use of Every Asset Class
    - From Highest Weight To Lowest Weight (Including Shorts) - Gold, US Bonds, Large-Cap Equities, Mid-Cap Equities, Small-Cap Equities, International Developed Markets, Emerging Markets, Bitcoin, Commodities
  - Long-Only (20% Target Return) - Makes Use of Just 3 Assets
    - US Bonds, Mid-Cap Equities, Gold

# Model Verification via Monte Carlo Simulation

- Goal was to check whether the simulated performance of each optimized portfolio aligns with the expected returns and variances predicted by the model.
- We use Monte Carlo Simulation to simulate future asset returns using the expected return vector and the covariance matrix
- We simulated 10,000 10-year return paths
- The portfolio weights are applied to the paths & their respected simulated performance is computed
- We find a Monte Carlo mean return and Monte Carlo volatility for both the unconstrained and constrained portfolios

# Sample Code for the Long-Only Monte Carlo Simulation. . .

```
n_paths = 10000
n_years = 10

# Extract shape of long-only weights matrix
n_assets, n_portfolios = long_only_weights_matrix.shape

np.random.seed(42)    # Ensure reproducibility

# Simulate Asset Returns
sim_asset_returns_lo = np.random.multivariate_normal(
    mean=mu,
    cov=sigma,
    size=(n_paths, n_years)
)

# Apply Long-Only Weights to Simulated Returns
sim_port_returns_lo = sim_asset_returns_lo @ long_only_weights_matrix

mc_mean_returns_lo = sim_port_returns_lo.mean(axis=(0, 1))
mc_vol_returns_lo  = sim_port_returns_lo.std(axis=(0, 1))

print("Long-Only MC Mean Returns:", mc_mean_returns_lo)
print()
print("Long-Only MC Volatilities:", mc_vol_returns_lo)

df_mc_longonly = pd.DataFrame({
    "Target Return": target_returns,
    "MC Mean Return": mc_mean_returns_lo,
    "MC Volatility": mc_vol_returns_lo
})

df_mc_longonly
```

# Results of Monte Carlo Simulation

Unconstrained Portfolios

	Target Return	MC Mean Return	MC Volatility
0	0.02	0.020097	0.050068
1	0.04	0.040104	0.050362
2	0.06	0.060110	0.054074
3	0.08	0.080117	0.060579
4	0.10	0.100124	0.069092
5	0.12	0.120131	0.078967
6	0.14	0.140138	0.089755
7	0.16	0.160145	0.101164
8	0.18	0.180151	0.113007
9	0.20	0.200158	0.125160

Constrained Portfolios

	Target Return	MC Mean Return	MC Volatility
0	0.02	0.020232	0.052843
1	0.04	0.040220	0.053621
2	0.06	0.060260	0.060567
3	0.08	0.080300	0.070689
4	0.10	0.100340	0.082830
5	0.12	0.120361	0.096270
6	0.14	0.140377	0.110596
7	0.16	0.160394	0.125506
8	0.18	0.180399	0.141014
9	0.20	0.200376	0.159737

# Final Notes & Next Steps

- Portfolio optimization relies on historical returns and covariances. These are inputs of how assets have behaved in the past and does not guarantee that assets will behave similarly in the future.
- What's important to take away is:
  - Unconstrained portfolios where shorting is allowed has less volatility against the constrained long-only portfolios
  - High returns (as shown in large-cap equities & Bitcoin) does not necessarily merit a high allocation. The assets' individual volatility and covariance are important considerations
- Next Steps:
  - We can perform the same analysis using a different historical period and then apply the model to an out-of-sample period to observe if the resulting model would be any different and if the model would perform as strongly when applied out-of sample.