Portfolio Construction with Views and Robust Covariance

This CQF capstone project implements a modular, numerical-stable, and further-customizable Black-Litterman model at its core. Views of assets can be provided directly as absolute views, relative views, or be estimated via qualitative views (e.g. bearish, bullish) along with market-implied volitility. 6 different methods are used to estimate the covariance matrix of assets' returns; and 8 different portfolio optimization methods are implemented to compare and to benchmark against the market in backtests.

Covariance matrix estimation methods used:

1. Empirical Estimator (Maximum Likelihood Estimator, biased)
2. Sample Estimator (unbiased)
3. Minimum Covariance Determinant Estimator (Rousseuw, 1984)
4. Linear Shrinkage Estimator (Ledoit and Wolf, 2004)
5. Non-linear Shrinkage Estimator (Ledoit and Wolf, 2017)
6. De-noising Estimator (Lopez de Prado, 2019)

Portfolio optimization methods used:

1. Maximum Sharpe ratio portfolio (MSRP), also known as the tangency portfolio and market portfolio
2. Mean-variance portfolio (MVP) with regularization
3. Global minimum variance portfolio (GMVP)
4. Maximum diversification ratio portfolio (MDRP)
5. Maximum decorrelation portfolio (MDP)
6. Risk parity portfolio (RPP), also known as the equal risk portfolio (ERP)
7. Minimum value at risk portfolio (MVaRP)
8. Minimum expected shortfall portfolio (MESP)

Along with the plots and analyses included, this project represents the delegate's own study and work. A list of references is also attached at the end of this report. The docstrings in each function defined follows NumPy style guide.

In [1]:

```
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
import pandas as pd
from scipy.optimize import minimize, root
from scipy.stats import norm
from sklearn.covariance import EmpiricalCovariance, LedoitWolf, MinCovDet
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KernelDensity
import seaborn as sns
import yfinance as yf
```

Portfolio Choice

The choice of portfolio assets must reflect optimal diversification. Similar to Idzorek's 2007 paper [6], we first form a portfolio by picking 6 large liquid ETFs that track **different** markets and asset classes (U.S. large-cap equities, international growth equities, international bonds, real estates, commodities, and Japan index). These assests are expected to be largely uncorrelated with one another.

1. Vanguard Large-Cap Index Fund ETF Shares (VV)
2. Vanguard Real Estate Index Fund ETF Shares (VNQ)
3. Vanguard International Growth Fund Investor Shares (VWIGX)
4. PIMCO International Bond Fund (U.S. Dollar-Hedged) Institutional Class (PFORX)
5. iShares S&P GSCI Commodity-Indexed Trust (GSG)
6. iShares MSCI Japan ETF (EWJ)

Note that if we want to allocate capital on a theme (e.g. for an industry, emerging market, fixed income assets, etc.), the specialized portfolio should have 5+ names, and 1-3 exogenous uncorrelated assets (e.g. commodity, long VIX, real estate, etc.)

We also form a second portfolio by picking 4 "optimal" stocks (bets) with strong personal views different from the market.

1. Alphabet Inc. (GOOGL) (U.S. Large-Cap Growth, 239.47% returns over the past 5 years)
2. Morgan Stanley (MS) (U.S. Large-Cap Value, 251.43% returns over the past 5 years)
3. The Blackstone Group Inc. (BX) (315.12% returns over the past 5 years)
4. ASML Holding N.V. (ASML) (Europe Developed, 622.46% over the past 5 years)

Note that the second portfolio bears the so-called **active risk**, as normally we should only hold something different than market weights if we are identifiably different than the market average investor. **This portfolio is picked solely for testing and comparison purpose throughout the project.**

We then compute the correlation matrix of the assets selected, and compare the rolling betas with respect to the market for each asset.

All price data are fetched from Yahoo Finance. For missing values due to different markets' open dates, we use forward fill to fill NaNs.

In [2]:

```
allocate_by_diversified_markets = True

if allocate_by_diversified_markets:
    assets_tickers = ['VV', 'VNQ', 'VWIGX', 'PFORX', 'GSG', 'EWJ']
else:
    assets_tickers = ['GOOGL', 'MS', 'BX', 'ASML']
```

```
market_ticker = 'QQQ' # to use as a strong market benchmark

assets_prices = yf.download(assets_tickers, start='2015-01-01', end='2021-12-31')['Adj Close'].fillna(method="ffill")
market_prices = yf.download(market_ticker, start='2015-01-01', end='2021-12-31')['Adj Close'].fillna(method="ffill")
```

```
[*********************100%***********************]  6 of 6 completed
[*********************100%***********************]  1 of 1 completed
```

In [3]:

```
pd.set_option('display.max_rows', None)
display(assets_prices.head(3))
print(assets_prices.shape)
```

Date	EWJ	GSG	PFORX	VNQ	VV	VWIGX
2015-01-02	40.739025	21.219999	8.660306	63.127037	83.444786	20.160070
2015-01-05	40.268684	20.620001	8.660306	63.472527	82.011429	19.719568
2015-01-06	39.617447	20.280001	8.676348	64.102097	81.215111	19.541492

```
(1664, 6)
```

In [4]:

```
display(market_prices.tail(3))
print(market_prices.shape)
```

```
Date
2021-08-09    368.730011
2021-08-10    366.839996
2021-08-11    366.209991
Name: Adj Close, dtype: float64
(1664,)
```

In [5]:

```
# assets' risks sorted in descending order
assets_prices.std().sort_values(ascending=False)
```

Out[5]:

```
VV       32.211946
VNQ      10.346590
VWIGX     9.748819
EWJ       7.483667
GSG       2.548643
PFORX     0.771312
dtype: float64
```

In [6]:

```
# correlation matrix of assets' returns
assets_prices.pct_change().dropna().corr()
```

Out[6]:

	EWJ	GSG	PFORX	VNQ	VV	VWIGX
EWJ	1.000000	0.325315	0.050115	0.575437	0.764814	0.688899
GSG	0.325315	1.000000	0.018941	0.259870	0.416335	0.371249
PFORX	0.050115	0.018941	1.000000	0.181167	0.078877	0.089515
VNQ	0.575437	0.259870	0.181167	1.000000	0.749167	0.514151
VV	0.764814	0.416335	0.078877	0.749167	1.000000	0.764511
VWIGX	0.688899	0.371249	0.089515	0.514151	0.764511	1.000000

In [7]:

```
def compute_rolling_beta(market_prices, asset_prices, window_len):
    """
    compute alpha and beta factors of assets given a rolling window length

    Parameters
    ----------
    market_prices : pd.Series
                    market prices
    asset_prices : pd.Series
                    asset prices
    window_len : bool
                    rolling window length

    Returns
    -------
    x : (np.ndarray, nd.ndarray)
        rolling alphas and betas of assets
    """
    asset_returns, market_returns = asset_prices.pct_change().dropna(), market_prices.pct_change().dropna()
    num_observations = len(asset_returns)
    alphas, betas = np.full(num_observations, np.nan), np.full(num_observations, np.nan)

    for i in range(num_observations - window_len):
        model = LinearRegression()
        model.fit(market_returns.values[i:i+window_len+1].reshape(-1, 1), asset_returns.values[i:i+window_len+1])
        alphas[i+window_len], betas[i+window_len] = model.intercept_, model.coef_[0]
    return alphas, betas
```

In [8]:

```
window_len = 30
assets_betas = pd.concat([pd.Series(compute_rolling_beta(market_prices, assets_prices[ticker], window_len)[1],
```

```
                                    index=assets_prices[ticker].index[1:],
                                    name=ticker) for ticker in assets_tickers], axis=1).dropna()
assets_betas.head()
```

Out[8]:

Date	VV	VNQ	VWIGX	PFORX	GSG	EWJ
2015-02-18	0.844864	0.265729	0.632282	-0.007408	0.400365	0.525857
2015-02-19	0.816875	0.287584	0.561174	-0.008114	0.285359	0.499339
2015-02-20	0.826066	0.374456	0.552334	0.001306	0.204894	0.440977
2015-02-23	0.822938	0.329285	0.554939	0.004668	0.233387	0.421345
2015-02-24	0.798059	0.350488	0.515112	-0.023583	0.241018	0.404759

In [9]:

```
# correlation matrix of assets' 30-day rolling betas
assets_betas.corr()
```

Out[9]:

	VV	VNQ	VWIGX	PFORX	GSG	EWJ
VV	1.000000	0.595585	0.199680	-0.201065	0.457429	0.543134
VNQ	0.595585	1.000000	0.058546	0.280602	0.273167	0.305762
VWIGX	0.199680	0.058546	1.000000	-0.060884	0.149671	0.371074
PFORX	-0.201065	0.280602	-0.060884	1.000000	-0.167814	-0.006391
GSG	0.457429	0.273167	0.149671	-0.167814	1.000000	0.172755
EWJ	0.543134	0.305762	0.371074	-0.006391	0.172755	1.000000

split historical and test data

Here we use use-specified lengths of "historical" and "test" daily-frequency trading data (fetched all the way back to 2015). Currently we plan to backtest the recent 30-day period using historical data that is several folds longer than the test data.

In [10]:

```
def standardize_returns_from_prices(prices):
    """
    standardize returns from prices
    the initial price is set to 1

    Parameters
    ----------
    prices : pd.Series
             ordered prices from past to recent

    Returns
    -------
    x : pd.Series
        standardized returns
    """
    returns = prices.pct_change()
    returns += 1
    returns.iloc[0] = 1 # set first day pseudo-price
    return returns.cumprod()

def generate_standardized_returns_data(num_test_days, num_historical_days, assets_tickers, assets_prices, market_prices):
    """
    generate standardized historical returns, backtest returns, and market returns

    Parameters
    ----------
    num_test_days : int
                    number of backtest days
    num_historical_days : int
                    number of historical days
    assets_tickers : list of str
                    assets' tickers
    assets_prices : pd.DataFrame
                    assets' prices in time series
    market_prices : pd.Series
                    market's prices in time series

    Returns
    -------
    x : (pd.DataFrame, pd.DataFrame, pd.Series)
        standardized historical returns, backtest returns, and market returns
    """
    returns_historical = pd.concat([standardize_returns_from_prices(assets_prices[ticker].iloc[-num_historical_days-num_test_days:-num_test_d
    returns_test = pd.concat([standardize_returns_from_prices(assets_prices[ticker].iloc[-num_test_days:]) for ticker in assets_tickers], axi
    market_returns_test = standardize_returns_from_prices(market_prices.iloc[-num_test_days:])
    return returns_historical, returns_test, market_returns_test
```

In [11]:

```
num_test_days = 30
num_historical_days = num_test_days * 4
returns_historical, returns_test, market_returns_test = generate_standardized_returns_data(num_test_days, num_historical_days, assets_tickers
```

```python
display(returns_historical.head(3))
display(returns_test.tail(3))

print('Historical "Training" Length (days):', num_historical_days)
print('Backtest Length (days):', num_test_days)
```

Date	VV	VNQ	VWIGX	PFORX	GSG	EWJ
2021-01-07	1.000000	1.000000	1.000000	1.0000	1.000000	1.000000
2021-01-08	1.006570	1.009724	1.027498	0.9982	1.014162	1.018372
2021-01-11	1.000112	0.996353	1.011619	0.9973	1.005507	1.008818

Date	VV	VNQ	VWIGX	PFORX	GSG	EWJ
2021-08-09	1.032677	1.044700	1.008585	1.01104	0.962710	1.003553
2021-08-10	1.032578	1.034777	1.025756	1.01104	0.980112	1.004146
2021-08-11	1.034720	1.040868	1.025756	1.01104	0.985705	1.016583

```
Historical "Training" Length (days): 120
Backtest Length (days): 30
```

Covariance Matrix Estimation

1. empirical covariance matrix estimator

The covariance matrix of a data set can also be estimated by the classical maximum likelihood estimator (or "empirical covariance matrix estimator"), provided the number of observations is large enough compared to the number of features (describing each observation).

Note that in scikit-learn's implementation, empirical covariance matrix estimator is biased with denominator N rather than N - 1.

In [12]:
```python
maximum_likelihood_estimator = EmpiricalCovariance().fit(returns_historical.values)
Σ_empirical = maximum_likelihood_estimator.covariance_
print('Empirical Estimation of Covariance Matrix:')
print(Σ_empirical)
```

```
Empirical Estimation of Covariance Matrix:
[[ 1.85808356e-03  3.34270438e-03 -1.74238871e-04 -1.43888047e-04
   2.79503277e-03 -1.12514711e-04]
 [ 3.34270438e-03  6.50643783e-03 -5.32008438e-04 -2.77266230e-04
   5.52924771e-03 -2.56291077e-04]
 [-1.74238871e-04 -5.32008438e-04  1.34400358e-03  1.22727414e-04
  -7.96063006e-04  3.51991260e-04]
 [-1.43888047e-04 -2.77266230e-04  1.22727414e-04  2.64676881e-05
  -2.80117203e-04  2.82826090e-05]
 [ 2.79503277e-03  5.52924771e-03 -7.96063006e-04 -2.80117203e-04
   5.64382833e-03 -3.22913160e-04]
 [-1.12514711e-04 -2.56291077e-04  3.51991260e-04  2.82826090e-05
  -3.22913160e-04  2.76355298e-04]]
```

2. sample covariance matrix estimator

The sample covariance matrix, which is unbiased, has N-1 in the denominator rather than N due to a variant of Bessel's correction. Note the numerical differences of `Σ_empirical` and `Σ_sample`.

In [13]:
```python
# reference: https://pandas.pydata.org/pandas-docs/dev/reference/api/pandas.DataFrame.cov.html
Σ_sample = returns_historical.cov().values
print('Sample Covariance Matrix:')
print(Σ_sample)

print('\nEmpirical Estimation of Covariance Matrix:')
print(returns_historical.cov(ddof=0).values) # same as EmpiricalCovariance()'s result
```

```
Sample Covariance Matrix:
[[ 1.87369771e-03  3.37079433e-03 -1.75703064e-04 -1.45097190e-04
   2.81852044e-03 -1.13460213e-04]
 [ 3.37079433e-03  6.56111378e-03 -5.36479098e-04 -2.79596199e-04
   5.57571198e-03 -2.58444784e-04]
 [-1.75703064e-04 -5.36479098e-04  1.35529772e-03  1.23758737e-04
  -8.02752611e-04  3.54949170e-04]
 [-1.45097190e-04 -2.79596199e-04  1.23758737e-04  2.66901056e-05
  -2.82471129e-04  2.85202780e-05]
 [ 2.81852044e-03  5.57571198e-03 -8.02752611e-04 -2.82471129e-04
   5.69125545e-03 -3.25626716e-04]
 [-1.13460213e-04 -2.58444784e-04  3.54949170e-04  2.85202780e-05
  -3.25626716e-04  2.78677612e-04]]
```


```
Empirical Estimation of Covariance Matrix:
[[ 1.85808356e-03  3.34270438e-03 -1.74238871e-04 -1.43888047e-04
   2.79503277e-03 -1.12514711e-04]
 [ 3.34270438e-03  6.50643783e-03 -5.32008438e-04 -2.77266230e-04
   5.52924771e-03 -2.56291077e-04]
 [-1.74238871e-04 -5.32008438e-04  1.34400358e-03  1.22727414e-04
  -7.96063006e-04  3.51991260e-04]
 [-1.43888047e-04 -2.77266230e-04  1.22727414e-04  2.64676881e-05
  -2.80117203e-04  2.82826090e-05]
 [ 2.79503277e-03  5.52924771e-03 -7.96063006e-04 -2.80117203e-04
   5.64382833e-03 -3.22913160e-04]
```

```
[-1.12514711e-04 -2.56291077e-04  3.51991260e-04  2.82826090e-05
  -3.22913160e-04  2.76355298e-04]]
```

3. linear shrinkage covariance matrix estimator

When the number of samples is small compared to the number of features, the empirical covariance matrix estimator is a poor estimator of the eigenvalues of the covariance matrix, so the inverse of the covariance matrix, often called the precision matrix, is not accurate. Sometimes, it even occurs that the empirical covariance matrix cannot be inverted for numerical reasons. To avoid such an inversion problem, the shrinkage, as a form of regularization is introduced to improve the estimation of covariance matrices.

Mathematically, the shrinkage consists in reducing the ratio between the smallest and the largest eigenvalues of the empirical covariance matrix. It can be done by simply shifting every eigenvalue according to a given offset, which is equivalent of finding the l2-penalized maximum likelihood estimator of the covariance matrix. In practice, shrinkage boils down to a simple a convex transformation:

$$\Sigma_{\text{shrunk}} = (1-\alpha)\hat{\Sigma} + \alpha \frac{\text{Tr}(\hat{\Sigma})}{\text{n}} \mathbf{1}$$

In their 2004 paper, Ledoit and Wolf proposed a formula to compute the optimal shrinkage coefficient that minimizes the mean squared error between the estimated and the real covariance matrices.

Note that the linear shrinkage covariance matrix estimator (Ledoit and Wolf, 2004) may not always be the best choice. For example if the distribution of the data is normally distributed, the Oracle Shrinkage Approximating (OSA) estimator yields a smaller mean squared error than the one given by Ledoit and Wolf's formula. [1]

In [14]:

```
linear_shrinkage_estimator = LedoitWolf().fit(returns_historical.values)
Σ_linear_shrinkage = linear_shrinkage_estimator.covariance_
print('\nLinear Srinkage Estimation of Covariance Matrix:')
print(Σ_linear_shrinkage)
```

```
Linear Srinkage Estimation of Covariance Matrix:
[[ 1.86720305e-03  3.30211957e-03 -1.72123384e-04 -1.42141058e-04
   2.76109742e-03 -1.11148636e-04]
 [ 3.30211957e-03  6.45912021e-03 -5.25549160e-04 -2.73899855e-04
   5.46211541e-03 -2.53179368e-04]
 [-1.72123384e-04 -5.25549160e-04  1.35936467e-03  1.21237343e-04
  -7.86397759e-04  3.47717625e-04]
 [-1.42141058e-04 -2.73899855e-04  1.21237343e-04  5.78253905e-05
  -2.76716214e-04  2.79392211e-05]
 [ 2.76109742e-03  5.46211541e-03 -7.86397759e-04 -2.76716214e-04
   5.60698392e-03 -3.18992572e-04]
 [-1.11148636e-04 -2.53179368e-04  3.47717625e-04  2.79392211e-05
  -3.18992572e-04  3.04679038e-04]]
```

robust covariance matrix estimator

Real data sets are often subject to measurement or recording errors. Regular but uncommon observations may also appear for a variety of reasons. Observations which are very uncommon are called outliers. The empirical covariance matrix estimator and the linear shrinkage covariance matrix estimators presented above are very sensitive to the presence of outliers in the data. Therefore, one should use robust covariance matrix estimators to estimate the covariance matrix of real data sets. On the other hand, robust covariance matrix estimators can be used to perform outlier detection and discard/downweight some observations according to further processing of the data.

4. minimum covariance determinant estimator

The minimum covariance determinant estimator is a robust estimator of a data set's covariance matrix introduced by Rousseeuw in 1984. The idea is to find a given proportion (h) of "good" observations which are not outliers and compute their empirical covariance matrix. This empirical covariance matrix is then rescaled to compensate the performed selection of observations ("consistency step"). Having computed the Minimum Covariance Determinant estimator, one can give weights to observations according to their Mahalanobis distance, leading to a reweighted estimate of the covariance matrix of the data set ("reweighting step").

In [15]:

```
minimum_covariance_determinant_estimator = MinCovDet(random_state=0).fit(returns_historical.values)
Σ_min_cov_det = minimum_covariance_determinant_estimator.covariance_
print('\nMinimum Covariance Determinant Estimation of Covariance Matrix:')
print(Σ_min_cov_det)
```

```
Minimum Covariance Determinant Estimation of Covariance Matrix:
[[ 1.88014214e-03  3.42262693e-03 -5.94270988e-04 -2.26187901e-04
   3.37738042e-03 -1.30234947e-04]
 [ 3.42262693e-03  6.66921849e-03 -1.13019575e-03 -4.09560164e-04
   6.53386143e-03 -3.25853022e-04]
 [-5.94270988e-04 -1.13019575e-03  1.03936250e-03  1.27834039e-04
  -1.29302040e-03  3.00190280e-04]
 [-2.26187901e-04 -4.09560164e-04  1.27834039e-04  3.29656994e-05
  -4.05635520e-04  3.44195511e-05]
 [ 3.37738042e-03  6.53386143e-03 -1.29302040e-03 -4.05635520e-04
   6.86263116e-03 -3.95562452e-04]
 [-1.30234947e-04 -3.25853022e-04  3.00190280e-04  3.44195511e-05
  -3.95562452e-04  2.13068308e-04]]
```

5. non-linear shrinkage covariance matrix estimator

With further improvement upon their 2004 paper, Ledoit and Wolf proposed a new non-linear shrinkage method in 2017 that does not require recovering the population eigenvalues first. This method estimates the sample spectral density and its Hilbert transform directly by smoothing the sample eigenvalues with a variable-bandwidth kernel. It can handle matrices of a dimension larger by a factor of ten. Even for dimension 10,000, the code runs in less than two minutes on a desktop computer; this makes the power of nonlinear shrinkage as accessible to applied statisticians as the one of linear shrinkage. The code below, provided in the project tutorial materials, is translated from the published MATLAB code along with the paper.

In [16]:

```
class DirectKernel(object):
    '''
```

```
    Ledoit, Olivier, and Michael Wolf. Direct nonlinear shrinkage estimation of large-dimensional covariance matrices. No. 264. Working Paper
    '''
    def __init__(self, X):
        self.X = X
        self.n = None
        self.p = None
        self.sample = None
        self.eigenvalues = None
        self.eigenvectors = None
        self.L = None
        self.h = None

    def pav(self, y):
        """
        PAV uses the pair adjacent violators method to produce a monotonic
        smoothing of y
        translated from matlab by Sean Collins (2006) as part of the EMAP toolbox
        """
        y = np.asarray(y)
        assert y.ndim == 1
        n_samples = len(y)
        v = y.copy()
        lvls = np.arange(n_samples)
        lvlsets = np.c_[lvls, lvls]
        flag = 1
        while flag:
            deriv = np.diff(v)
            if np.all(deriv >= 0):
                break

            viol = np.where(deriv < 0)[0]
            start = lvlsets[viol[0], 0]
            last = lvlsets[viol[0] + 1, 1]
            s = 0
            n = last - start + 1
            for i in range(start, last + 1):
                s += v[i]

            val = s / n
            for i in range(start, last + 1):
                v[i] = val
                lvlsets[i, 0] = start
                lvlsets[i, 1] = last
        return v

    def estimate_cov_matrix(self):
        # extract sample eigenvalues sorted in ascending order and eigenvectors
        self.n, self.p = self.X.shape
        self.sample = (self.X.T @ self.X) / self.n
        self.eigenvalues, self.eigenvectors = np.linalg.eig(self.sample)
        isort = np.argsort(self.eigenvalues, axis=-1)
        self.eigenvalues.sort()
        self.eigenvectors = self.eigenvectors[:, isort]

        # compute direct kernel estimator
        self.eigenvalues = self.eigenvalues[max(1, self.p - self.n + 1) - 1:self.p]
        self.L = np.repeat(self.eigenvalues, min(self.n, self.p), axis=0).reshape(self.eigenvalues.shape[0], min(self.n, self.p))
        self.h = self.n ** (-0.35)
        component_00 = 4*(self.L.T**2)*self.h**2 - (self.L - self.L.T)**2
        component_0 = np.maximum(np.zeros((component_00.shape[1], component_00.shape[1])), component_00)
        component_a = np.sqrt(component_0)
        component_b = 2*np.pi*(self.L.T**2)*self.h**2
        ftilda = np.mean(component_a / component_b, axis=1)

        com_1 = np.sign(self.L - self.L.T)
        com_2_1 = (self.L - self.L.T)**2 - 4*self.L.T**2*self.h**2
        com_2 = np.maximum(np.zeros((com_2_1.shape[1], com_2_1.shape[1])), com_2_1)
        com_3_1 = np.sqrt(com_2)
        com_3_2 = com_1 * com_3_1
        com_3 = com_3_2 - self.L + self.L.T
        com_4 = 2*np.pi*self.L.T**2*self.h**2
        com_5 = com_3 / com_4
        Hftilda = np.mean(com_5, axis=1)

        if self.p <= self.n:
            com_0 = (np.pi*(self.p/self.n)*self.eigenvalues*ftilda)**2
            com_1 = (1 - (self.p / self.n) - np.pi * (self.p / self.n) * self.eigenvalues * Hftilda) ** 2
            com_2 = com_0 + com_1
            dtilde = self.eigenvalues / com_2
        else:
            Hftilda0 = (1-np.sqrt(max(1-4*self.h**2, 0))) / (2*np.pi*self.n*self.h**2)*np.mean(1/self.eigenvalues)
            dtilde0 = 1/(np.pi*((self.p-self.n)/self.n)*Hftilda0)
            dtilde1 = self.eigenvalues/np.pi**2*self.eigenvalues**2*(ftilda**2+Hftilda**2)
            dtilde = np.hstack((dtilde0*np.ones((self.p-self.n, 1)).reshape(self.p-self.n,), dtilde1))

        dhat = self.pav(dtilde)
        d_matrix = np.diag(dhat)
        sigmahat = self.eigenvectors.dot(d_matrix).dot(self.eigenvectors.T)
        return sigmahat
```

In [17]:

```
direct_nonlinear_shrinkage_estimator = DirectKernel(returns_historical.values)
```

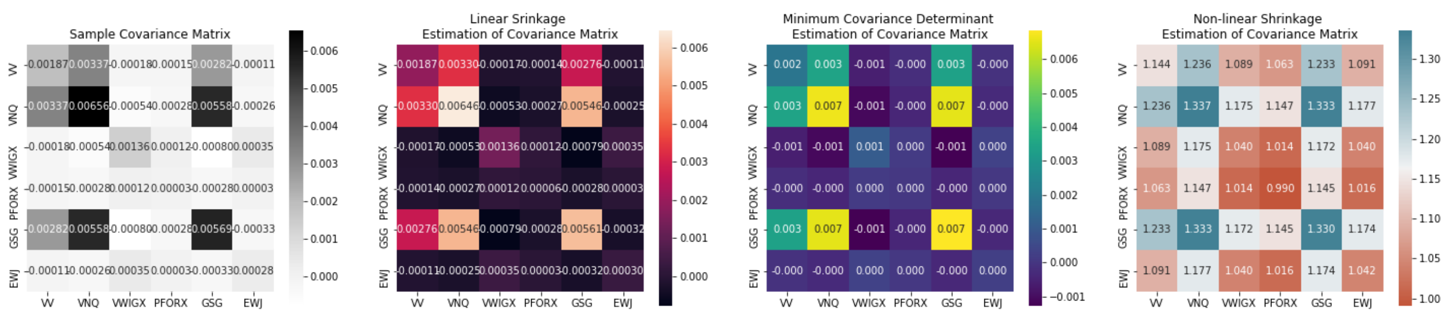
```
Σ_non_linear_shrinkage = direct_nonlinear_shrinkage_estimator.estimate_cov_matrix()
print('\nNon-linear Shrinkage Estimation of Covariance Matrix:')
print(Σ_non_linear_shrinkage)
```

```
Non-linear Shrinkage Estimation of Covariance Matrix:
[[1.14401199 1.23582109 1.08886917 1.06315445 1.23285004 1.09063973]
 [1.23582109 1.3365116  1.17460245 1.14708821 1.33290764 1.17672449]
 [1.08886917 1.17460245 1.03987693 1.01404836 1.17205814 1.04045862]
 [1.06315445 1.14708821 1.01404836 0.98998252 1.14483033 1.01553115]
 [1.23285004 1.33290764 1.17205814 1.14483033 1.33044295 1.17435013]
 [1.09063973 1.17672449 1.04045862 1.01553115 1.17435013 1.0420076 ]]
```

In [18]:

```
fig, ax = plt.subplots(nrows=1, ncols=4, figsize=(25,5))
ax[0].set_title('Sample Covariance Matrix')
sns.heatmap(Σ_sample,
            xticklabels=assets_tickers,
            yticklabels=assets_tickers,
            cmap='Greys',
            annot=True,
            fmt=".5f",
            square=True,
            ax=ax[0])
ax[1].set_title('Linear Srinkage \nEstimation of Covariance Matrix')
sns.heatmap(Σ_linear_shrinkage,
            xticklabels=assets_tickers,
            yticklabels=assets_tickers,
            annot=True,
            fmt=".5f",
            square=True,
            ax=ax[1])
ax[2].set_title('Minimum Covariance Determinant \nEstimation of Covariance Matrix')
sns.heatmap(Σ_min_cov_det,
            xticklabels=assets_tickers,
            yticklabels=assets_tickers,
            cmap='viridis',
            annot=True,
            fmt=".3f",
            square=True,
            ax=ax[2])
ax[3].set_title('Non-linear Shrinkage \nEstimation of Covariance Matrix')
sns.heatmap(Σ_non_linear_shrinkage,
            xticklabels=assets_tickers,
            yticklabels=assets_tickers,
            cmap=sns.diverging_palette(20, 220, n=200),
            annot=True,
            fmt=".3f",
            square=True,
            ax=ax[3])
```

Out[18]: <AxesSubplot:title={'center':'Non-linear Shrinkage \nEstimation of Covariance Matrix'}>

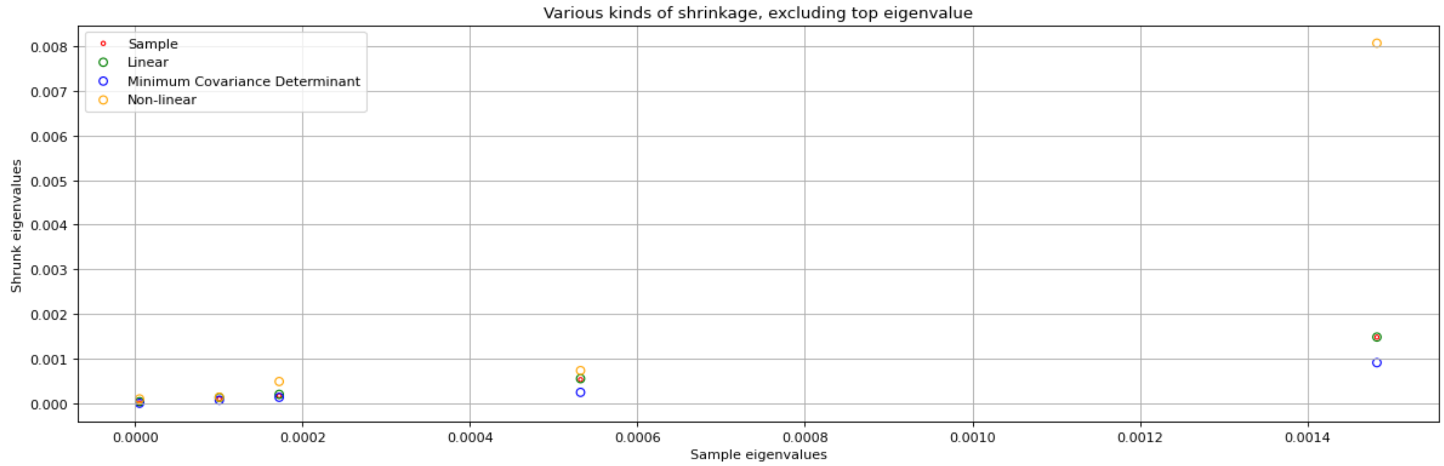


In [19]:

```
eigenvalues_sample, _ = np.linalg.eig(Σ_sample)
eigenvalues_linear, _ = np.linalg.eig(Σ_linear_shrinkage)
eigenvalues_min_cov_det, _ = np.linalg.eig(Σ_min_cov_det)
eigenvalues_non_linear_shrinkage, _ = np.linalg.eig(Σ_non_linear_shrinkage)

eigenvalues_sample.sort()
eigenvalues_linear.sort()
eigenvalues_min_cov_det.sort()
eigenvalues_non_linear_shrinkage.sort()
```

In [20]:

```
fig = plt.figure(figsize=(15,5), dpi=80, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)
ax.scatter(eigenvalues_sample[:-1], eigenvalues_sample[:-1], marker='.', facecolors='none', edgecolors='red', label='Sample')
ax.scatter(eigenvalues_sample[:-1], eigenvalues_linear[:-1], marker='o', facecolors='none', edgecolors='green', label='Linear')
ax.scatter(eigenvalues_sample[:-1], eigenvalues_min_cov_det[:-1], marker='o', facecolors='none', edgecolors='blue', label='Minimum Covariance
ax.scatter(eigenvalues_sample[:-1], eigenvalues_non_linear_shrinkage[:-1], marker='o', facecolors='none', edgecolors='orange', label='Non-lin
ax.set(xlabel='Sample eigenvalues', ylabel='Shrunk eigenvalues', title='Various kinds of shrinkage, excluding top eigenvalue')
plt.legend()
plt.grid()
```



From the above plot, we see that the sample eigenvalues of linear shrinkage estimation and sample estimation of covariance matrix are not much different. However, the sample eigenvalues of non-linear estimation of covariance matrix differ significantly from the ones from other estimation methods.

6. de-noising estimator

In Lopez de Prado's 2019 paper [13], a de-nosing method used for the estimation of covariance matrix of assets returns. In short, de-noising essentially replaces the eigenvalues below the Marchenko-Pastur threshold with their average. As a result, the smallest eigenvalue becomes bigger, while the "signal" eigenvalues remain untouched, thus contributing less noise in matrix inversion used in portfolio optimizations. De-noising is arguably better than the shrinkage method based on the test results included in its paper.

Consider a matrix of independent and identically distributed random observations X, of size T x N, where the underlying process generating the observations has zero mean and variance σ^2. The matrix $C = T^{-1}X'X$ has eigenvalues λ that asymptotically converge (as $N \to +\infty$ and $T \to +\infty$ with $1 < \frac{T}{N} < +\infty$) to the Marcenko-Pastur probability density function (PDF):

$$f[\lambda] = \begin{cases} \frac{T}{N}\frac{\sqrt{(\lambda_+ - \lambda)(\lambda - \lambda_-)}}{2\pi\lambda\sigma^2} & \text{if } \lambda \in [\lambda_-, \lambda_+] \\ 0 & \text{if } \lambda \notin [\lambda_-, \lambda_+] \end{cases}$$

where the maximum expected eigenvalue is $\lambda_+ = \sigma^2\left(1 + \sqrt{\frac{N}{T}}\right)^2$, and the minimum expected eigenvalue is $\lambda_- = \sigma^2\left(1 - \sqrt{\frac{N}{T}}\right)^2$.

Eigenvalues $\lambda \in [\lambda_-, \lambda_+]$ are consistent with random behavior, and eigenvalues $\lambda \notin [\lambda_-, \lambda_+]$ are consistent with non-random behavior. Specifically, we associate eigenvalues $\lambda \in [0, \lambda_+]$ with noise.

Let $\{\lambda_n\}_{n=1,\dots,N}$ be the set of all eigenvalues, ordered descending, and i be the position of the eigenvalue such that $\lambda_i > \lambda_+$ and $\lambda_{i+1} \leq \lambda_+$.

Then we set $\lambda_j = \frac{1}{N}\sum_{k=i+1}^{N}\lambda_k, \quad j = i+1, \dots, N$ hence preserving the trace of the correlation matrix.

Given the eigenvector decomposition $VW = V\Lambda$, we form the de-noised correlation matrix C as

$$\tilde{C}_1 = W\tilde{\Lambda}W'$$
$$C_1 = \tilde{C}_1\left[\left(\operatorname{diag}\left[\tilde{C}_1\right]\right)^{1/2}\left(\operatorname{diag}\left[\tilde{C}_1\right]\right)^{1/2'}\right]^{-1}$$

where Λ is the diagonal matrix holding the corrected eigenvalues. The reason for the second transformation is to re-scale the matrix $\tilde{C}_1$, so that the main diagonal of C_1 is an array of 1s. [13]

The following code (with minor adaptation) is directly from Lopez de Prado's paper with all author's original comments retained. Note that the maximum random eigenvector is found by fitting Marchenko-Pastur's distribution to the empirical kernal density estimation (see function `findMaxEval`) and correlation matrices are flattened for values greater than 1 or smaller than -1 (see function `cov2corr`).

In [21]:
```
def fitKDE(obs, bWidth=.25, kernel='gaussian', x=None):
    # Fit kernel to a series of obs, and derive the prob of obs
    # x is the array of values on which the fit KDE will be evaluated
    if len(obs.shape) == 1:
        obs = obs.reshape(-1, 1)
    kde = KernelDensity(kernel=kernel, bandwidth=bWidth).fit(obs)
    if x is None:
        x = np.unique(obs).reshape(-1, 1)
    if len(x.shape) == 1:
        x = x.reshape(-1, 1)
    logProb = kde.score_samples(x)  # log(density)
    pdf = pd.Series(np.exp(logProb), index=x.flatten())
    return pdf

def mpPDF(var, q, pts):
    # Marcenko-Pastur pdf
    # q=T/N
    (eMin, eMax) = (var * (1 - (1. / q) ** .5) ** 2, var * (1 + (1. / q) ** .5) ** 2)
    eVal = np.linspace(eMin, eMax, pts)
    pdf = q / (2 * np.pi * var * eVal) * ((eMax - eVal) * (eVal - eMin)) ** .5
    pdf = pd.Series(pdf.flatten(), index=eVal.flatten())
```

```
    return pdf

def errPDFs(var, eVal, q, bWidth, pts=1000):
    # Fit error
    pdf0 = mpPDF(var, q, pts)  # theoretical pdf
    pdf1 = fitKDE(eVal, bWidth, x=pdf0.index.values)  # empirical pdf
    sse = np.sum((pdf1 - pdf0) ** 2)
    return sse

def findMaxEval(eVal, q, bWidth):
    # Find max random eVal by fitting Marcenko's dist to the empirical one
    out = minimize(lambda *x: errPDFs(*x), .5, args=(eVal, q, bWidth), bounds=((1E-5, 1 - 1E-5), ))
    if out['success']:
        var = out['x'][0]
    else:
        var = 1
    eMax = var * (1 + (1. / q) ** .5) ** 2
    return (eMax, var)

def corr2cov(corr, std):
    cov = corr * np.outer(std, std)
    return cov

def cov2corr(cov):
    # Derive the correlation matrix from a covariance matrix
    std = np.sqrt(np.diag(cov))
    corr = cov / np.outer(std, std)
    (corr[corr < -1], corr[corr > 1]) = (-1, 1)  # numerical error
    return corr

def getPCA(matrix):
    # Get eVal,eVec from a Hermitian matrix
    (eVal, eVec) = np.linalg.eigh(matrix)
    indices = eVal.argsort()[::-1]  # arguments for sorting eVal desc
    (eVal, eVec) = (eVal[indices], eVec[:, indices])
    eVal = np.diagflat(eVal)
    return (eVal, eVec)

def denoisedCorr(eVal, eVec, nFacts):
    # Remove noise from corr by fixing random eigenvalues
    eVal_ = np.diag(eVal).copy()
    eVal_[nFacts:] = eVal_[nFacts:].sum() / float(eVal_.shape[0] - nFacts)
    eVal_ = np.diag(eVal_)
    corr1 = np.dot(eVec, eVal_).dot(eVec.T)
    corr1 = cov2corr(corr1)
    return corr1

def deNoiseCov(cov0, q, bWidth):
    corr0 = cov2corr(cov0)
    (eVal0, eVec0) = getPCA(corr0)
    (eMax0, var0) = findMaxEval(np.diag(eVal0), q, bWidth)
    nFacts0 = eVal0.shape[0] - np.diag(eVal0)[::-1].searchsorted(eMax0)
    print ('Number of preserved non-random eigenvector:', nFacts0)
    corr1 = denoisedCorr(eVal0, eVec0, nFacts0)
    cov1 = corr2cov(corr1, np.diag(cov0) ** .5)
    return cov1
```

In [22]:

```
q = returns_historical.shape[0]/returns_historical.shape[1]
Σ_de_noised = deNoiseCov(returns_historical.cov().values,q,0.25)
Σ_de_noised
```

```
Number of preserved non-random eigenvector: 1
```

Out[22]:

```
array([[ 1.87369771e-03,  2.11718111e-03, -6.96678950e-04,
        -1.32237041e-04,  1.98229544e-03, -2.79778550e-04],
       [ 2.11718111e-03,  6.56111378e-03, -1.32145317e-03,
        -2.50825803e-04,  3.75999677e-03, -5.30680959e-04],
       [-6.96678950e-04, -1.32145317e-03,  1.35529772e-03,
         8.25366597e-05, -1.23726335e-03,  1.74625709e-04],
       [-1.32237041e-04, -2.50825803e-04,  8.25366597e-05,
         2.66901056e-05, -2.34845683e-04,  3.31458084e-05],
       [ 1.98229544e-03,  3.75999677e-03, -1.23726335e-03,
        -2.34845683e-04,  5.69125545e-03, -4.96871259e-04],
       [-2.79778550e-04, -5.30680959e-04,  1.74625709e-04,
         3.31458084e-05, -4.96871259e-04,  2.78677612e-04]])
```

The Black-Litterman Model

The Black-Litterman model (BL) provides a framework in which more satisfactory results could be obtained from a larger set of inputs: view portfolios, the expected returns on those portfolios, the confidence in the view portfolios and the uncertainty on the reference model. Using BL, a portfolio manager could process those inputs, blend them into the reference return distribution, and obtain an optimal allocation that reflected the views in a consistent way without corner solutions.

In [23]:

```
%%time
def generate_marketcap_weights(assets_tickers, allocate_by_diversified_markets):
    """
    generate assets' weights by their market capitalizations

    Parameters
    ----------
    assets_tickers : list
                     list of assets' tickers
```

```
    allocate_by_diversified_markets : bool
                    whether assets by diversified by markets or classes

    Returns
    -------
    x : np.ndarray
        assets' weights by their market capitalization
    """
    key = 'totalAssets' if allocate_by_diversified_markets else 'marketCap'
    try:
        asset_marketcap_dict = dict()
        for ticker in assets_tickers:
            asset_marketcap_dict[ticker] = yf.Ticker(ticker).info[key]
    except Exception as error: # in case Yahoo Finance API does not load ETF prices properly
        print('Key error! Use market capitalization values instead.')
        asset_marketcap_dict = {'VV': 37649666048, 'VNQ': 77342482432, 'VWIGX': 72672477184, 'PFORX': 13071910912, 'GSG': 1.355*10**9, 'EWJ':

    asset_marketcaps = pd.Series(asset_marketcap_dict)
    marketcap_weights = asset_marketcaps / asset_marketcaps.sum()
    print(f"assets' market capitalization weights:\n{marketcap_weights}")
    return marketcap_weights.values.reshape(-1, 1)

w_market = generate_marketcap_weights(assets_tickers, allocate_by_diversified_markets)
```

```
assets' market capitalization weights:
VV       0.176187
VNQ      0.361935
VWIGX    0.340081
PFORX    0.061172
GSG      0.006396
EWJ      0.054231
dtype: float64
CPU times: user 943 ms, sys: 97 ms, total: 1.04 s
Wall time: 15.1 s
```

In [24]:

```
risk_free_rate = 0.0007 # use 1-year Treasury rate 0.07%
τ = 1/len(returns_historical)
# λ = 2.24 # an empirical value for risk aversion
# use a market Sharpe ratio of 0.5 (the same as Black and Litterman), Σ_non_linear_shrinkage is used as an example
λ = 0.5/np.sqrt(w_market.T @ Σ_non_linear_shrinkage @ w_market).flatten()[0]
```

The main feature of the Black-Litterman model is that it incorporate the views of analysts and portfolio managers to fine tune our estimates of expected returns and our asset allocation.

- Each view has a corresponding row in the picking matrix (the order matters)
- Absolute views have a single 1 in the column corresponding to the ticker's order in the universe.
- Relative views have a positive number in the nominally outperforming asset columns and a negative number in the nominally underperforming asset columns. The numbers in each row should sum up to 0.

If the user has only qualitative views, we can use estimate views on assets' returns by setting entries of v in terms of the volatility induced by the market:

$$v_k \equiv (\mathbf{P}\boldsymbol{\pi})_k + \eta_k \sqrt{(\mathbf{P}\boldsymbol{\Sigma}\mathbf{P}')_{k,k}}, \quad k = 1, \ldots, K$$

where $\eta_k \in \{-\beta, -\alpha, +\alpha, +\beta\}$ defines "very bearish", "bearish", "bullish" and "very bullish" views respectively for the kth asset. Typical choices for these parameters are $\alpha \equiv 1$ and $\beta \equiv 2$. [7]

Note that we can also applied supervised learning to multinomially predict/classify the qualitative views based on historical returns and features (EMA, RSI, MACD, etc.) derived historical prices.

In [25]:

```
def generate_views(λ, Σ, w_market, P, η):
    """
    estimate views on assets' returns from qualitative views in terms of market-implied volatility

    Parameters
    ----------
    λ : float
        risk aversion coefficient that characterizes the expected risk-return tradeoff
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)
    w_market :
        the market allocations (K x 1 column vector)
    P : np.ndarray
        the picking matrix that identifies the assets involved in the views (K x N matrix)
    η : np.ndarray
        qualitative views vector (K x 1 column vector)

    Returns
    -------
    x : np.ndarray
        estimated views on assets' returns
    """
    Π = λ*Σ @ w_market
    return P @ Π + np.diag(np.sqrt(P @ Σ @ P.T)).reshape(-1, 1)*η
```

In the following example, we use absolute views on the assets in our portfolio.

In [26]:

```
'''
Portfolio 1 of diversified assets:
```

```
1. Vanguard Large-Cap Index Fund ETF Shares (VV)                              absolute return: 0.3
2. Vanguard Real Estate Index Fund ETF Shares (VNQ)                           absolute return: 0.03
3. Vanguard International Growth Fund Investor Shares (VWIGX)                 absolute return: 0.3
4. PIMCO International Bond Fund (U.S. Dollar-Hedged) Institutional Class (PFORX)  absolute return: 0.005
5. iShares S&P GSCI Commodity-Indexed Trust (GSG)                             absolute return: 0.02
6. iShares MSCI Japan ETF (EWJ)                                               absolute return: 0.06

Portfolio 2 of hand-pciked stocks with very strong views:
1. Alphabet Inc. (GOOGL)                         (U.S. Large-Cap Growth)  absolute return: 0.5
2. Morgan Stanley (MS)                           (U.S. Large-Cap Value)   absolate return: 0.45
3. The Blackstone Group Inc. (BX)                                         absolute return: 0.6
4. ASML Holding N.V. (ASML)                      (Europe Developed)       absolute return: 1
'''
η = np.ones((len(assets_tickers), 1)) # bullish on all assets (as an example when we construct the portfolio)

if allocate_by_diversified_markets:
    P = np.eye(len(assets_tickers))
    Q = np.array([0.3, 0.03, 0.3, 0.005, 0.02, 0.06]).reshape(-1, 1)
#     Q = generate_views(λ, Σ, w_market, P, η)
else:
    P = np.eye(len(assets_tickers))
    Q = np.array([0.5, 0.45, 0.6, 1]).reshape(-1, 1)
#     Q = generate_views(λ, Σ, w_market, P, η)

print(f'τ: {τ:.6f}')
print(f'λ: {λ:.3f}')
```

```
τ: 0.008333
λ: 0.464
```

Meucci [7] shows how to obtain the distribution of μ_{BL} and Σ_{BL} given the views using Bayes' formula:

$$\boldsymbol{\mu}_{BL} \equiv \left((\tau\boldsymbol{\Sigma})^{-1} + \mathbf{P}'\boldsymbol{\Omega}^{-1}\mathbf{P}\right)^{-1}\left((\tau\boldsymbol{\Sigma})^{-1}\boldsymbol{\pi} + \mathbf{P}'\boldsymbol{\Omega}^{-1}\mathbf{v}\right)$$

$$\boldsymbol{\Sigma}_{BL} \equiv \Sigma + \left((\tau\boldsymbol{\Sigma})^{-1} + \mathbf{P}'\boldsymbol{\Omega}^{-1}\mathbf{P}\right)^{-1}$$

which can be rearranged to an equivalent and computationally more stable representations:

$$\boldsymbol{\mu}_{BL} = \boldsymbol{\pi} + \tau\boldsymbol{\Sigma}\mathbf{P}'\left(\tau\mathbf{P}\boldsymbol{\Sigma}\mathbf{P}' + \boldsymbol{\Omega}\right)^{-1}(\mathbf{v} - \mathbf{P}\boldsymbol{\pi})$$
$$\boldsymbol{\Sigma}_{BL} = (1+\tau)\boldsymbol{\Sigma} - \tau^2\boldsymbol{\Sigma}\mathbf{P}'\left(\tau\mathbf{P}\boldsymbol{\Sigma}\mathbf{P}' + \boldsymbol{\Omega}\right)^{-1}\mathbf{P}\boldsymbol{\Sigma}$$

In [27]:

```
def black_litterman(λ, Σ, w_market, τ, Q, P):
    """
    The Black-Litterman model takes a Bayesian approach to asset allocation.
    Specifically, it combines a prior estimate of returns (for example, the market-implied returns) with views on certain assets,
    to produce a posterior estimate of expected excess returns.

    Parameters
    ----------
    λ : float
        risk aversion coefficient that characterizes the expected risk-return tradeoff
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)
    w_market :
        the market allocations (K x 1 column vector)
    τ : float
        a scalar to tune the standard error of estimate for the equilibrium vector Π
    Q : np.ndarray
        the views vector (K x 1 column vector)
    P : np.ndarray
        the picking matrix that identifies the assets involved in the views (K x N matrix)

    Returns
    -------
    x : (np.ndarray, np.ndarray)
        posterior estimate of expected excess returns, and its variances
    """
    Π = λ*Σ @ w_market # implied equilibrium vector of excess return
    Ω = np.diag(np.diag(P @ (τ*Σ) @ P.T)) # approach proposed by He and Litterman [8]
    μ_BL = Π + τ*Σ @ P.T @ inv(τ*P @ Σ @ P.T + Ω) @ (Q - P @ Π) # computationally stable representations of μ_BL and Σ_BL
    Σ_BL = (1 + τ)*Σ - τ*τ*Σ @ P.T @ inv(τ*P @ Σ @ P.T + Ω) @ P @ Σ
    return μ_BL, Σ_BL

μ_BL_sample, Σ_BL_sample = black_litterman(λ, Σ_sample, w_market, τ, Q, P)
μ_BL_linear_shrinkage, _ = black_litterman(λ, Σ_linear_shrinkage, w_market, τ, Q, P)
μ_BL_min_cov_det, _ = black_litterman(λ, Σ_min_cov_det, w_market, τ, Q, P)
μ_BL_non_linear_shrinkage, _ = black_litterman(λ, Σ_non_linear_shrinkage, w_market, τ, Q, P)
μ_BL_de_noised, _ = black_litterman(λ, Σ_de_noised, w_market, τ, Q, P)

print('μ_BL_sample:\n', μ_BL_sample)
print('\nμ_BL_linear_shrinkage:\n', μ_BL_linear_shrinkage)
print('\nμ_BL_min_cov_det:\n', μ_BL_min_cov_det)
print('\nμ_BL_non_linear_shrinkage:\n', μ_BL_non_linear_shrinkage)
print('\nμ_BL_de_noised:\n', μ_BL_de_noised)
```

```
μ_BL_sample:
 [[0.09965851]
 [0.14108125]
 [0.15617233]
```

```
 [0.00551377]
 [0.06594928]
 [0.04600117]]

μ_BL_linear_shrinkage:
 [[0.10071517]
 [0.13739708]
 [0.1561217 ]
 [0.00533225]
 [0.0641668 ]
 [0.04575655]]

μ_BL_min_cov_det:
 [[0.06382778]
 [0.0976588 ]
 [0.13412316]
 [0.00260446]
 [0.06226235]
 [0.04988978]]

μ_BL_non_linear_shrinkage:
 [[0.18064704]
 [0.19491396]
 [0.17253127]
 [0.16809807]
 [0.19430582]
 [0.17249515]]

μ_BL_de_noised:
 [[0.08541506]
 [0.00677993]
 [0.12194602]
 [0.00157911]
 [0.00372689]
 [0.02588577]]
```

maximizing mean-variance trade-off

In [28]:

```python
def minimize_mean_variance_tradeoff(λ, μ, Σ):
    """
    mean-variance portfolio optimization without constaint

    Parameters
    ----------
    λ : float
        risk aversion coefficient
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    return 1/λ*(inv(Σ) @ μ)
```

In [29]:

```python
w_BL_mean_var_tradeoff = minimize_mean_variance_tradeoff(λ, μ_BL_non_linear_shrinkage, Σ_non_linear_shrinkage) # use Σ_non_linear_shrinkage a
w_BL_mean_var_tradeoff
```

Out[29]:

```
array([[ 0.40079529],
       [ 0.09628535],
       [ 0.60398434],
       [-0.2935151 ],
       [-0.27566306],
       [-0.17819627]])
```

return vectors and resulting portfolio weights

In [30]:

```python
μ_BL = μ_BL_non_linear_shrinkage # use μ_BL_non_linear_shrinkage and Σ_non_linear_shrinkage as an example
Σ = Σ_non_linear_shrinkage

analysis = {'Asset': assets_tickers,
            'New Combined Return Vector E(R)': μ_BL.flatten(),
            'Implied Equilibrium Return Vector Π': (λ*Σ @ w_market).flatten(),
            'Difference E(R) - Π': (μ_BL - λ*Σ @ w_market).flatten(),
            'New Weight w (no constraint)': w_BL_mean_var_tradeoff.flatten(),
            'Market Capitalization Weight w_mkt': w_market.flatten(),
            'Difference w - w_mkt': (w_BL_mean_var_tradeoff - w_market).flatten()
           }
pd.DataFrame(analysis).round(decimals=4)
```

Out[30]:

	Asset	New Combined Return Vector E(R)	Implied Equilibrium Return Vector Π	Difference E(R) - Π	New Weight w (no constraint)	Market Capitalization Weight w_mkt	Difference w - w_mkt
0	VV	0.1806	0.5347	-0.3541	0.4008	0.1762	0.2246
1	VNQ	0.1949	0.5776	-0.3826	0.0963	0.3619	-0.2656
2	VWIGX	0.1725	0.5093	-0.3368	0.6040	0.3401	0.2639
3	PFORX	0.1681	0.4971	-0.3290	-0.2935	0.0612	-0.3547
4	GSG	0.1943	0.5762	-0.3819	-0.2757	0.0064	-0.2821

	Asset	New Combined Return Vector E(R)	Implied Equilibrium Return Vector Π	Difference E(R) - Π	New Weight w (no constraint)	Market Capitalization Weight w_mkt	Difference w - w_mkt
5	EWJ	0.1725	0.5100	-0.3375	-0.1782	0.0542	-0.2324

The New Weight (w) in column 5 of table above is based on the New Combined Return Vector E(R). One of the strongest features of the Black-Litterman model is illustrated in the final column. Only the weights of the assets for which views were expressed changed from their original market capitalization weights and the directions of the changes are intuitive.

From a macro perspective, the new portfolio can be viewed as the sum of two portfolios, where Portfolio 1 is the original market capitalization-weighted portfolio, and Portfolio 2 is a series of long and short positions based on the views. Portfolio 2 can be subdivided into mini-portfolios, each associated with a specific view. The relative views result in mini-portfolios with offsetting long and short positions that sum to 0.

One can fine tune the Black-Litterman model by studying the New Combined Return Vector E(R), calculating the anticipated risk-return characteristics of the new portfolio and then adjusting the scalar (τ) and the individual variances of the error term (ω) that form the diagonal elements of the covariance matrix of the error term (Ω). [5]

Portfolio Allocation

To determine which covariance matrix estimation method for assets' returns to use; and which posterior estimate of assets' expected excess returns to use, we use maximum Sharpe ratio portfolio as the benchmark. The parameters pair with the best performance will be selected to use in the following different optimization methods.

1. maximum Sharpe ratio portfolio (MSRP), also known as the tangency portfolio and market portfolio

1.1 maximizing Sharpe ratio with budget constraint

$$\begin{array}{ll} \underset{\mathbf{w}}{\text{maximize}} & \dfrac{\mathbf{w}^T\boldsymbol{\mu} - r_f}{\sqrt{\mathbf{w}^T\boldsymbol{\Sigma}\mathbf{w}}} \\ \text{subject to} & \mathbf{1}^T\mathbf{w} = 1 \end{array}$$

This is a convex quadratic problem (QP) with the following closed-form solution:

$$\mathbf{w}_{\text{MSRP}} = \frac{\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - r_f\mathbf{1})}{\mathbf{1}^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} - r_f\mathbf{1})}$$

In addition to budget and no short position constraints, we can also add the following constraints in practice:

$$\begin{array}{ll} \|\mathbf{w}\|_1 \leq \gamma & \text{leverage} \\ \|\mathbf{w} - \mathbf{w}_0\|_1 \leq \tau & \text{turnover} \\ \|\mathbf{w}\|_\infty \leq u & \text{max position} \\ \|\mathbf{w}\|_0 \leq K & \text{sparsity} \end{array}$$

In [31]:

```
def maximize_sharpe_ratio_with_budget(r, μ, Σ):
    """
    maximum Sharpe ratio portfolio optimization with budget constraint

    Parameters
    ----------
    r : float
        risk-free rate
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    u = np.ones((len(μ), 1))
    return inv(Σ) @ (μ - r*u) / (u.T @ inv(Σ) @ (μ - r*u))
```

In [32]:

```
w_BL_max_sharpe_ratio = maximize_sharpe_ratio_with_budget(risk_free_rate, μ_BL_non_linear_shrinkage, Σ_non_linear_shrinkage)
w_BL_max_sharpe_ratio
```

Out[32]:

```
array([[ 0.78763182],
       [ 0.48187979],
       [ 1.86754093],
       [-1.3271398 ],
       [-0.68580268],
       [-0.12411006]])
```

1.2 maximizing Sharpe ratio with budget constraint and no short position

$$\begin{array}{ll} \mathbf{w}^T\mathbf{1} = 1 & \text{budget} \\ \mathbf{w} \geq \mathbf{0} & \text{no short position} \end{array}$$

In [33]:

```
def maximize_sharpe_ratio_with_constraints(r, μ, Σ):
    """
    maximum Sharpe ratio portfolio optimization with budget constraint and no short position
```

```
    Parameters
    ----------
    r : float
        risk-free rate
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    objective = lambda w: -(w.T @ μ - r) / np.sqrt(w.T @ Σ @ w) # negative Sharpe ratio
    bounds = tuple((0, 1) for bound in range(len(μ))) # no short position
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    w_init = np.ones((len(μ), 1))/len(μ) # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [34]:

```
w_BL_max_sharpe_ratio_with_constraints = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL, Σ)
w_BL_max_sharpe_ratio_with_constraints
```

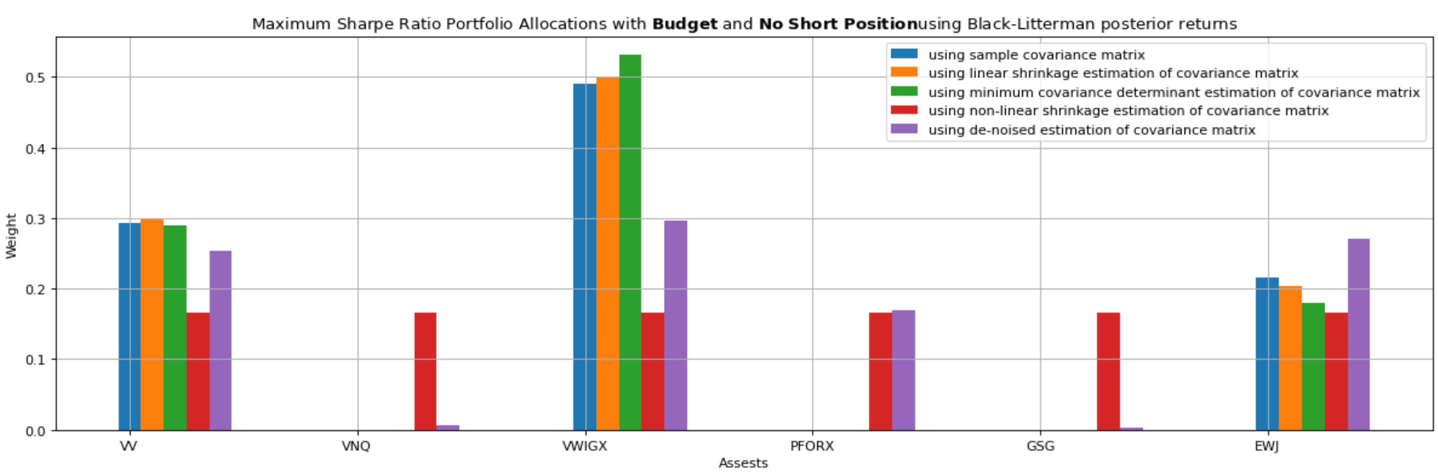
Out[34]:

```
array([[0.16666667],
       [0.16666667],
       [0.16666667],
       [0.16666667],
       [0.16666667],
       [0.16666667]])
```

In the bar char below, we compare the maximum Sharpe ratio portfolio allocations by views from the Black-Litterman model using different estimations of covariances matrix.

In [35]:

```
w_BL_max_sharpe_ratio_with_constraints_Σ_sample = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL_sample, Σ_sample)
w_BL_max_sharpe_ratio_with_constraints_Σ_linear_shrinkage = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL_linear_shrinkage, Σ_l
w_BL_max_sharpe_ratio_with_constraints_Σ_min_cov_det = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL_min_cov_det, Σ_min_cov_det
w_BL_max_sharpe_ratio_with_constraints_Σ_de_noised = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL_de_noised, Σ_de_noised)
w_BL_max_sharpe_ratio_with_constraints = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL_non_linear_shrinkage, Σ_non_linear_shrin

fig = plt.figure(figsize=(15,5), dpi=80, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)
ax.bar(assets_tickers, w_BL_max_sharpe_ratio_with_constraints_Σ_sample.flatten(), width=0.1,
       label='using sample covariance matrix')
ax.bar(np.arange(len(assets_tickers))+0.1, w_BL_max_sharpe_ratio_with_constraints_Σ_linear_shrinkage.flatten(), width=0.1,
       label='using linear shrinkage estimation of covariance matrix')
ax.bar(np.arange(len(assets_tickers))+0.2, w_BL_max_sharpe_ratio_with_constraints_Σ_min_cov_det.flatten(), width=0.1,
       label='using minimum covariance determinant estimation of covariance matrix')
ax.bar(np.arange(len(assets_tickers))+0.3, w_BL_max_sharpe_ratio_with_constraints.flatten(), width=0.1,
       label='using non-linear shrinkage estimation of covariance matrix')
ax.bar(np.arange(len(assets_tickers))+0.4, w_BL_max_sharpe_ratio_with_constraints_Σ_de_noised.flatten(), width=0.1,
       label='using de-noised estimation of covariance matrix')

ax.set(xlabel='Assests', ylabel='Weight', title=r'Maximum Sharpe Ratio Portfolio Allocations with $\bf{Budget}$ and $\bf{No}$ $\bf{Short}$ $\
       'using Black-Litterman posterior returns')
ax.legend()
ax.grid()
```

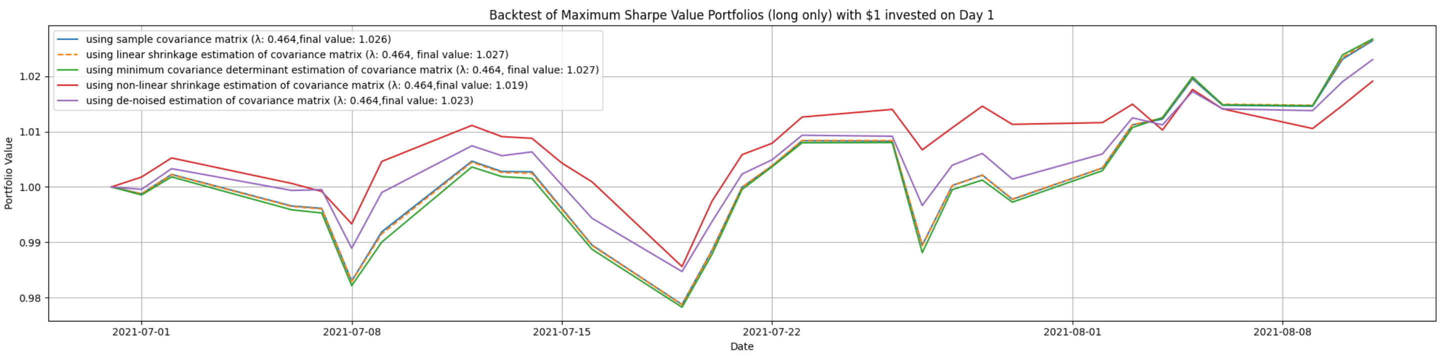


To see which estimation method has better performance, in the plot below we compare different covariance matrix estimation methods in a backtest of the recent test trading days. We see that non-linear shrinkage and de-noising deliver the best results. **Interestingly we notice that non-linear shrinkage seems to be consistently better than de-noising throughout our backtest period, which technically should be better than non-linear shrinkage method. This result is due to the low number of assets picked in the portfolio, as currently only 1 eigenvector is kept as non-random contribution to covariance estimation. As number of assets gets larger (N > 10), de-noising should be much better than non-linear shrinkage method.**

In [36]:

```
fig = plt.figure(figsize=(20,5), dpi=100, tight_layout=True)
```

```python
ax = fig.add_subplot(1, 1, 1)
ax.plot(returns_test @ w_BL_max_sharpe_ratio_with_constraints_Σ_sample, \
        label=f'using sample covariance matrix (λ: {λ:.3f},' \
        f'final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio_with_constraints_Σ_sample)[0], 3)})')
ax.plot(returns_test @ w_BL_max_sharpe_ratio_with_constraints_Σ_linear_shrinkage, linestyle='--', \
        label=f'using linear shrinkage estimation of covariance matrix (λ: {λ:.3f},' \
        f' final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio_with_constraints_Σ_linear_shrinkage)[0], 3)})')
ax.plot(returns_test @ w_BL_max_sharpe_ratio_with_constraints_Σ_min_cov_det, \
        label=f'using minimum covariance determinant estimation of covariance matrix (λ: {λ:.3f},' \
        f' final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio_with_constraints_Σ_min_cov_det)[0], 3)})')
ax.plot(returns_test @ w_BL_max_sharpe_ratio_with_constraints, \
        label=f'using non-linear shrinkage estimation of covariance matrix (λ: {λ:.3f},' \
        f'final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio_with_constraints)[0], 3)})')
ax.plot(returns_test @ w_BL_max_sharpe_ratio_with_constraints_Σ_de_noised, \
        label=f'using de-noised estimation of covariance matrix (λ: {λ:.3f},' \
        f'final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio_with_constraints_Σ_de_noised)[0], 3)})')
ax.set(xlabel='Date', ylabel='Portfolio Value', title='Backtest of Maximum Sharpe Value Portfolios (long only) with $1 invested on Day 1')
ax.legend()
ax.grid()
```



In [37]:
```python
μ_BL = μ_BL_non_linear_shrinkage # chosen pair of parameters to use in the following portfolio optimizations
Σ = Σ_non_linear_shrinkage
```

2. mean-variance portfolio (MVP)

2.1 maximizing mean-variance trade-off with budget constraint

$$\begin{aligned} \underset{\mathbf{w}}{\text{maximize}} \quad & \mathbf{w}^T\boldsymbol{\mu} - \lambda\mathbf{w}^T\boldsymbol{\Sigma}\mathbf{w} \\ \text{subject to} \quad & \mathbf{1}^T\mathbf{w} = 1 \end{aligned}$$

This is a convex quadratic problem (QP) with the following closed-form solution:

$$\mathbf{w}_{\text{MVP}} = \frac{1}{2\lambda}\boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu} + \nu\mathbf{1})$$

where ν is the optimal dual variable $\nu = \frac{2\lambda - \mathbf{1}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}}{\mathbf{1}^T\boldsymbol{\Sigma}^{-1}\mathbf{1}}$.

In [38]:
```python
def maximize_mean_variance_tradeoff_with_budget(λ, μ, Σ):
    """
    mean-variance portfolio optimization with budget constraint

    Parameters
    ----------
    λ : float
        risk aversion coefficient
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    u = np.ones((Σ.shape[0], 1))
    ν = (2*λ - u.T @ inv(Σ) @ μ) / (u.T @ inv(Σ) @ u)
    return 1/(2*λ)*(inv(Σ) @ (μ + ν*u))
```

In [39]:
```python
w_BL_mean_var_tradeoff = maximize_mean_variance_tradeoff_with_budget(λ, μ_BL, Σ)
w_BL_mean_var_tradeoff
```

Out[39]:
```
array([[ 8.38724296],
       [-4.12891004],
       [-1.64862854],
       [ 9.60937951],
       [-2.74406828],
       [-8.47501562]])
```

2.2 maximizing mean-variance trade-off with budget constraint and no short position

$$\begin{aligned} \mathbf{w}^T\mathbf{1} &= 1 \quad \text{budget} \\ \mathbf{w} &\geq \mathbf{0} \quad \text{no short position} \end{aligned}$$

In [40]:

```
def maximize_mean_variance_tradeoff_with_constraints(λ, μ, Σ, regularize=False, γ=1):
    """
    mean-variance portfolio optimization with budget constraint

    Parameters
    ----------
    λ : float
        risk aversion coefficient
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)
    regularize: bool
        whether to further diversify mean-variance portfolio if allocations are concentrated
    γ : float
        hyperparameter to tune the strength of regularization

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    if regularize:
        objective = lambda w: -(w.T @ μ - λ*(w.T @ Σ @ w)) + γ*(w.T @ w)
    else:
        objective = lambda w: -(w.T @ μ - λ*(w.T @ Σ @ w))
    bounds = tuple((0, 1) for bound in range(len(μ))) # no short position
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    w_init = np.ones((len(μ), 1))/len(μ) # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [41]:

```
w_BL_mean_var_tradeoff_with_constraints = maximize_mean_variance_tradeoff_with_constraints(λ, μ_BL, Σ)
w_BL_mean_var_tradeoff_with_constraints
```

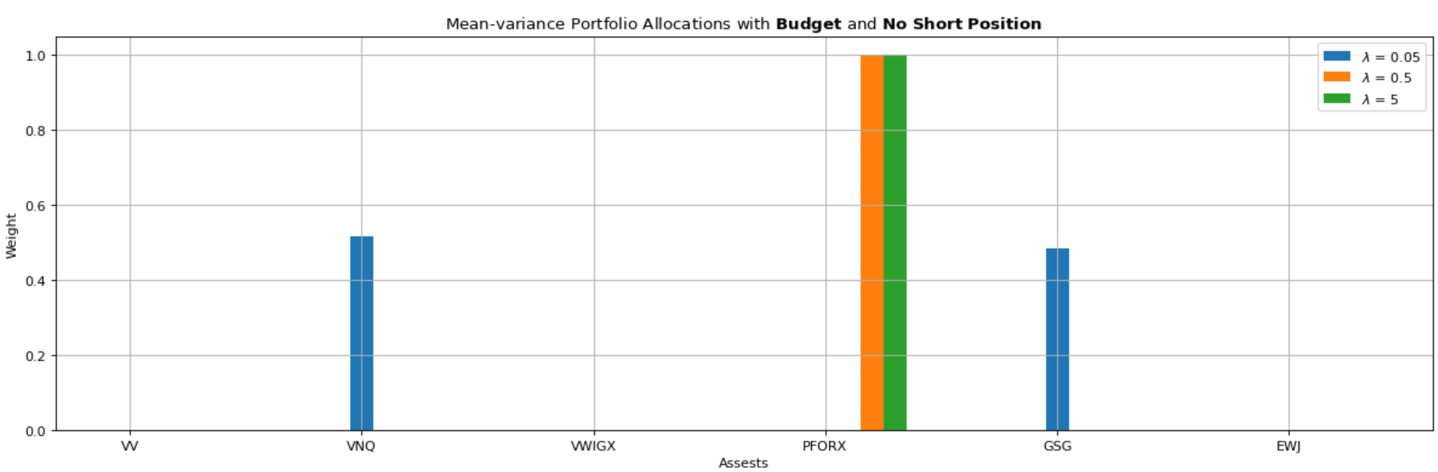
Out[41]:

```
array([[0.00000000e+00],
       [5.43363556e-17],
       [1.76941795e-16],
       [1.00000000e+00],
       [2.64676368e-16],
       [0.00000000e+00]])
```

In the bar char below, we compare the mean-variance portfolio allocations using different risk aversion coefficients. We expect to see that with lower risk aversion, the optimized portfolio allocates capital to higher return and higher risk assets, On the other hand, with the higher risk aversion, the optimized portfolio would allocate capital to lower return and lower risk assets. However, often times we hit the so-called "corner problem" of mean-variance optimization in which case allocation is highly concentrated in just a couple of assets setting many other assets' weights to zero, as the example below shows.

In [42]:

```
fig = plt.figure(figsize=(15,5), dpi=80, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)

w_1 = maximize_mean_variance_tradeoff_with_constraints(0.05, μ_BL, Σ)
w_2 = maximize_mean_variance_tradeoff_with_constraints(0.5, μ_BL, Σ)
w_3 = maximize_mean_variance_tradeoff_with_constraints(5, μ_BL, Σ)

ax.bar(assets_tickers, w_1.flatten(), width=0.1, label=r'$\lambda$ = 0.05')
ax.bar(np.arange(len(assets_tickers))+0.2, w_2.flatten(), width=0.1, label=r'$\lambda$ = 0.5')
ax.bar(np.arange(len(assets_tickers))+0.3, w_3.flatten(), width=0.1, label=r'$\lambda$ = 5')
ax.set(xlabel='Assests', ylabel='Weight', title='Mean-variance Portfolio Allocations' + r' with $\bf{Budget}$ and $\bf{No}$ $\bf{Short}$ $\bf
ax.legend()
ax.grid()
```



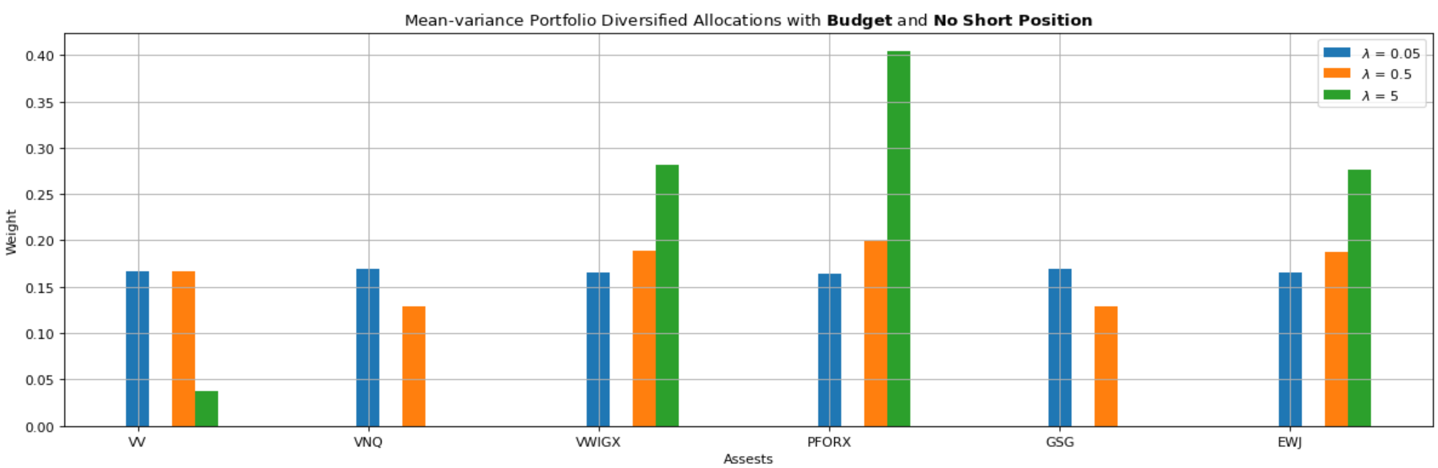
To combat this issue, we can borrow the idea of regularization from machine learning. Essentially, by adding an additional cost function to the objective, we "encourage" the optimizer to choose a "more diversified" portfolio. Chamberlain [1983] proposes the sum of squared weights or L_2 norm. Green and Hollifield [1992], interested in the relationship between diversification and the asset number N, declare that the portfolio is diversified if every asset is below a threshold weight of K(N). Bouchard [1997] proposes the L_P norm, which means the sum of the p-power of the weights, and, as limit case, the entropy of weights. [9]

Diversification Regularizer $D(w)$	Author
$D = \sum_i w_i^2$	Chamberlain [1983]
$D = \max [w_i]$	Green [1992]
$D = \sum_i w_i^P$	Bouchard [1997]
$D = \sum_i w_i \ln(w_i)$	Bouchard [1997]
$D = \prod_i w_i$	Corvalán [2005]

In the bar chart below, we used L_2 norm to further diversify the mean-variance portfolio. The result aligns with our intuition that with lower risk aversion, the optimized portfolio allocates more capital to riskier assets with higher returns, and with the higher risk aversion, the optimized portfolio would allocate more capital to less risky assets with lower returns.

In [43]:

```
fig = plt.figure(figsize=(15,5), dpi=80, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)

w_1 = maximize_mean_variance_tradeoff_with_constraints(0.05, μ_BL, Σ, regularize=True)
w_2 = maximize_mean_variance_tradeoff_with_constraints(0.5, μ_BL, Σ, regularize=True)
w_3 = maximize_mean_variance_tradeoff_with_constraints(5, μ_BL, Σ, regularize=True)

ax.bar(assets_tickers, w_1.flatten(), width=0.1, label=r'$\lambda$ = 0.05')
ax.bar(np.arange(len(assets_tickers))+0.2, w_2.flatten(), width=0.1, label=r'$\lambda$ = 0.5')
ax.bar(np.arange(len(assets_tickers))+0.3, w_3.flatten(), width=0.1, label=r'$\lambda$ = 5')
ax.set(xlabel='Assests', ylabel='Weight', title='Mean-variance Portfolio Diversified Allocations' + r' with $\bf{Budget}$ and $\bf{No}$ $\bf{
ax.legend()
ax.grid()
```



In summary, there are 4 major drawbacks of the mean-variance (MV) optimizer:

1. The first drawback is lack of diversification of optimal portfolios, as aforementioned "corner problem". MV optimizer tends to heavily weigh those asset classes that show high estimated returns compared to low variances (or standard deviations) and negative correlations. These are also the asset classes that are most likely to suffer from large estimation errors. For this reason, Michaud (1989) wrote: "The unintuitive character of many optimized portfolios can be traced to the fact that MV optimizers are, in a fundamental sense, estimation-error maximizers." [11]
2. The second drawback is instability of optimal portfolios. This attribute defines the high sensitivity of portfolio allocations to small changes in the estimated parameters. Especially, in the case we have in the investment universe couples of asset classes with similar risk-return profile, small perturbations may completely alter the distribution of the portfolio weights because they may cause an alternation between the dominant and the dominated asset class. [11]
3. The third drawback consists of failing to recognise the non-uniqueness of optimal portfolios. As sharply noted by Michaud (1989): "Optimizers, in general, produce, a unique optimal portfolio for a given level of risk. This appearance of exactness is highly misleading, however. The uniqueness of the solution depends on the erroneous assumption that the inputs are without statistical estimation error." Hence, in practice, it would be helpful, in order to reach the goal of identifying recommended portfolio structures, to consider many statistically equivalent portfolios and take the average weights resulting from their different compositions. [11]
4. The last but potentially the most serious drawback affects MV optimizer's poor out-of-sample performance: Outside the sample period used to estimate input parameters, classical Markowitz's portfolios show a considerable deterioration of performance with respect to the expected one and the same occurs in terms of risk-adjusted performance. As observed by DeMiguel et al. (2009) and Jobson and Korkie (1981), the extent to which they fall short of the original targeted performance is such that very simple investment criteria can dominate MV Optimization. Thus, this means that the 'plug-in rule' cannot be validated. [11] We'll see later in the backtesting section that MV optimization indeed underperforms some other portfolio optimizations using out-of-sample data.

In [44]:

```
# use the further diversified mean-variance portfolio weights in the following cells
w_BL_mean_var_tradeoff_with_constraints = maximize_mean_variance_tradeoff_with_constraints(λ, μ_BL, Σ, regularize=True)
```

3. global minimum variance portfolio (GMVP)

3.1 minimizing variance with budget constraint

$$\begin{array}{ll} \underset{\mathbf{w}}{\text{minimize}} & \mathbf{w}^T \mathbf{\Sigma} \mathbf{w} \\ \text{subject to} & \mathbf{1}^T \mathbf{w} = 1 \end{array}$$

This is a convex quadratic problem (QP) with the following closed-form solution:

$$\mathbf{w}_{\mathsf{GMVP}} = \frac{\mathbf{\Sigma}^{-1}\mathbf{1}}{\mathbf{1}^T\mathbf{\Sigma}^{-1}\mathbf{1}}$$

In [45]:

```python
def minimize_variance_with_budget(Σ):
    """
    global minimum variance portfolio optimization with budget constraint

    Parameters
    ----------
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    u = np.ones((Σ.shape[0], 1))
    return inv(Σ) @ u / (u.T @ inv(Σ) @ u)
```

In [46]:

```python
w_min_var = minimize_variance_with_budget(Σ)
w_min_var
```

Out[46]:

```
array([[  9.94569435],
       [ -5.07444418],
       [ -2.36968901],
       [ 11.85213027],
       [ -3.16615657],
       [-10.18753487]])
```

3.2 minimizing variance with budget constraint and no short position

$$\begin{aligned} \mathbf{w}^T\mathbf{1} &= 1 \quad && \text{budget} \\ \mathbf{w} &\geq \mathbf{0} && \text{no short position} \end{aligned}$$

In [47]:

```python
def minimize_variance_with_constraints(Σ):
    """
    global minimum variance portfolio optimization with budget constraint and no short position

    Parameters
    ----------
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    objective = lambda w: w.T @ Σ @ w
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    bounds = tuple((0, 1) for bound in range(Σ.shape[0])) # no short position
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [48]:

```python
w_min_var_with_constraints = minimize_variance_with_constraints(Σ)
w_min_var_with_constraints
```

Out[48]:

```
array([[0.00000000e+00],
       [4.68681430e-16],
       [0.00000000e+00],
       [1.00000000e+00],
       [4.57852086e-16],
       [0.00000000e+00]])
```

4. most diversified portfolio (MDP)

The diversification ratio (DR) is defined analogous to the Sharpe ratio (SR):

$$\mathsf{DR} = \frac{\mathbf{w}^T\boldsymbol{\sigma}}{\sqrt{\mathbf{w}^T\mathbf{\Sigma}\mathbf{w}}}$$

where $\boldsymbol{\sigma}^2 = \mathsf{Diag}(\mathbf{\Sigma})$

$$\begin{array}{ll} \underset{\mathbf{w}}{\mathsf{maximize}} & \dfrac{\mathbf{w}^T\boldsymbol{\sigma}}{\sqrt{\mathbf{w}^T\mathbf{\Sigma}\mathbf{w}}} \\ \mathsf{subject\ to} & \mathbf{1}^T\mathbf{w} = 1. \end{array}$$

In [49]:

```python
def maximize_diversification_ratio_with_constraints(Σ):
    """
    maximum diversification ratio portfolio optimization with budget constraint and no short position
```

```
    Parameters
    ----------
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    σ = np.sqrt(np.diag(Σ))
    objective = lambda w: -(w.T @ σ) / np.sqrt(w.T @ Σ @ w) # negative diversification ratio
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    bounds = tuple((0, 1) for bound in range(Σ.shape[0])) # no short position
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [50]:

```
w_BL_max_diversification_ratio_with_constraints = maximize_diversification_ratio_with_constraints(Σ)
w_BL_max_diversification_ratio_with_constraints
```

Out[50]:

```
array([[0.16666667],
       [0.16666667],
       [0.16666667],
       [0.16666667],
       [0.16666667],
       [0.16666667]])
```

5. maximum decorrelation portfolio (MDCP)

The maximum decorrelation portfolio (MDCP) is defined to minimize assets' correlation. MDCP is closely related to GMVP and MDP, but applies to the case where an investor believes all assets have similar returns and volatility, but heterogeneous correlations.

$$\begin{array}{lll} \underset{\mathbf{w}}{\text{minimize}} & \mathbf{w}^T\mathbf{C}\mathbf{w} & \\ \text{subject to} & \mathbf{1}^T\mathbf{w} = 1 & \text{budget} \\ & \mathbf{w} \geq \mathbf{0} & \text{no short position} \end{array}$$

where correlation matrix $\mathbf{C} \triangleq \text{Diag}(\mathbf{\Sigma})^{-1/2}\mathbf{\Sigma}\text{Diag}(\mathbf{\Sigma})^{-1/2}$

In [51]:

```
def maximize_decorrelation_with_constraints(Σ):
    """
    maximum decorrelation portfolio optimization with budget constraint and no short position

    Parameters
    ----------
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    C = np.sqrt(np.diag(np.diag(Σ))) @ Σ @ np.sqrt(np.diag(np.diag(Σ)))
    objective = lambda w: w.T @ C @ w
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    bounds = tuple((0, 1) for bound in range(Σ.shape[0])) # no short position
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [52]:

```
w_BL_max_decorrelation_with_constraints = maximize_decorrelation_with_constraints(Σ)
w_BL_max_decorrelation_with_constraints
```

Out[52]:

```
array([[4.07921987e-16],
       [3.89571169e-16],
       [0.00000000e+00],
       [1.00000000e+00],
       [9.40712418e-16],
       [0.00000000e+00]])
```

6. risk parity portfolio (RPP), also known as the equal risk portfolio (ERP)

Recall that risk contribution (RC) of the ith asset is defined as the follows:

$$\text{RC}_i = w_i\frac{\partial\sigma}{\partial w_i} = \frac{w_i(\mathbf{\Sigma w})_i}{\sqrt{\mathbf{w}^T\mathbf{\Sigma w}}}$$

In risk parity portfolio (RPP):

$$\text{RC}_i = \frac{w_i(\mathbf{\Sigma w})_i}{\sqrt{\mathbf{w}^T\mathbf{\Sigma w}}} = b_i\sigma(\mathbf{w}) = b_i\sqrt{\mathbf{w}^T\mathbf{\Sigma w}}$$

where risk budget $b_i = \frac{1}{N}$ is the same for each asset class.

By rearranging the risk contribution expression, we obtain a system of non-linear equations:

$$w_i(\mathbf{\Sigma w})_i = b_i \mathbf{w}^T \mathbf{\Sigma w} = \frac{1}{N}\mathbf{w}^T \mathbf{\Sigma w}, \qquad i = 1, \ldots, N$$

To write this system of non-linear equations more compactly, we can define $\mathbf{x} = \mathbf{w}/\sqrt{\mathbf{w}^T \mathbf{\Sigma w}}$ and have:

$$\mathbf{\Sigma x} = (\frac{1}{N})/\mathbf{x}$$

In [53]:

```
def risk_parity(Σ):
    """
    risk parity portfolio optimization without constraint

    Parameters
    ----------
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    b = np.ones(Σ.shape[0])/Σ.shape[0]
    objective = lambda w: Σ @ (w/np.sqrt(w.T @ Σ @ w)) - b/(w/np.sqrt(w.T @ Σ @ w))
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = root(objective, w_init, method='lm') # solve a system of non-linear risk budget equations
    return result['x'].reshape(-1,1)
```

In [54]:

```
w_risk_parity = risk_parity(Σ)
w_risk_parity
```

Out[54]:

```
array([[53.55092448],
       [49.58797353],
       [56.21402923],
       [57.58457599],
       [49.69650398],
       [56.13081187]])
```

To add constraints, we can also form this system of non-linear equations to the following optimization problem:

$$\begin{array}{lll} \underset{\mathbf{w}}{\text{minimize}} & \sum_i^N (w_i(\mathbf{\Sigma w})_i - \frac{1}{N}\mathbf{w}^T\mathbf{\Sigma w})^2 & \\ \text{subject to} & \mathbf{1}^T\mathbf{w} = 1 & \text{budget} \\ & \mathbf{w} \geq \mathbf{0} & \text{no short position} \end{array}$$

In [55]:

```
def risk_parity_with_constraints(Σ):
    """
    risk parity portfolio optimization with budget constraint and no short position

    Parameters
    ----------
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    b = np.ones(Σ.shape[0])/Σ.shape[0]
    objective = lambda w: np.sum(np.square(w * (Σ @ w) - b*(w.T @ Σ @ w)))
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    bounds = tuple((0, 1) for bound in range(Σ.shape[0])) # no short position
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [56]:

```
w_risk_parity_with_constraints = risk_parity_with_constraints(Σ)
w_risk_parity_with_constraints
```

Out[56]:

```
array([[0.1658719 ],
       [0.15357798],
       [0.17424547],
       [0.17842084],
       [0.15389589],
       [0.17398793]])
```

7. minimum value at risk portfolio (MVaRP) and expected shortfall portfolio (MESP)

Assuming portfolio return is normally distributed, we minimize portfolio's value at risk (VaR) and expected shortfall (ES) as follows [12]:

$$\begin{array}{lll} \underset{\mathbf{w}}{\text{minimize}} & -\mathbf{w}^T\mu + \text{Factor} \times \sqrt{\mathbf{w}^T\mathbf{\Sigma w}} & \\ \text{subject to} & \mathbf{1}^T\mathbf{w} = 1 & \text{budget} \\ & \mathbf{w} \geq \mathbf{0} & \text{no short position} \end{array}$$

where $\Pr\{\text{Loss}(\text{w}) \leq \text{VaR}_\alpha(\text{w})\} = \alpha$
VaR factor is $\Phi^{-1}(\alpha)$
ES factor is $\frac{1}{1-\alpha}\phi(\Phi^{-1}(\alpha))$

In [57]:
```
def minimize_value_at_risk_with_constraints(μ, Σ, α):
    """
    minimize value at risk with budget constraint and no short position

    Parameters
    ----------
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)
    α : float
        the probability such that Pr{L(𝑥)≤VaR𝑐(𝑥)}=𝛼

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    factor = norm.ppf(α)
    objective = lambda w: -w.T @ μ + factor*np.sqrt(w.T @ Σ @ w)
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    bounds = tuple((0, 1) for bound in range(Σ.shape[0])) # no short position
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [58]:
```
def minimize_expected_shortfall_with_constraints(μ, Σ, α):
    """
    minimize expected shortfall with budget constraint and no short position

    Parameters
    ----------
    μ : np.ndarray
        expected returns of assets
    Σ : np.ndarray
        the covariance matrix of excess returns (N x N matrix)
    α : float
        the probability such that Pr{L(𝑥)≤VaR𝑐(𝑥)}=𝛼

    Returns
    -------
    x : np.ndarray
        assets allocation weights
    """
    factor = 1/(1 - α)*norm.pdf(norm.ppf(α))
    objective = lambda w: -w.T @ μ + factor*np.sqrt(w.T @ Σ @ w)
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1}) # budget
    bounds = tuple((0, 1) for bound in range(Σ.shape[0])) # no short position
    w_init = np.ones((Σ.shape[0], 1))/Σ.shape[0] # initial weights guess
    result = minimize(objective, w_init, method='SLSQP', bounds=bounds, constraints=constraints)
    return result['x'].reshape(-1, 1)
```

In [59]:
```
w_BL_min_VaR_with_constraints = minimize_value_at_risk_with_constraints(μ_BL, Σ, α=0.95)
w_BL_min_VaR_with_constraints
```

Out[59]:
```
array([[2.64720506e-16],
       [0.00000000e+00],
       [0.00000000e+00],
       [1.00000000e+00],
       [2.66830051e-16],
       [0.00000000e+00]])
```

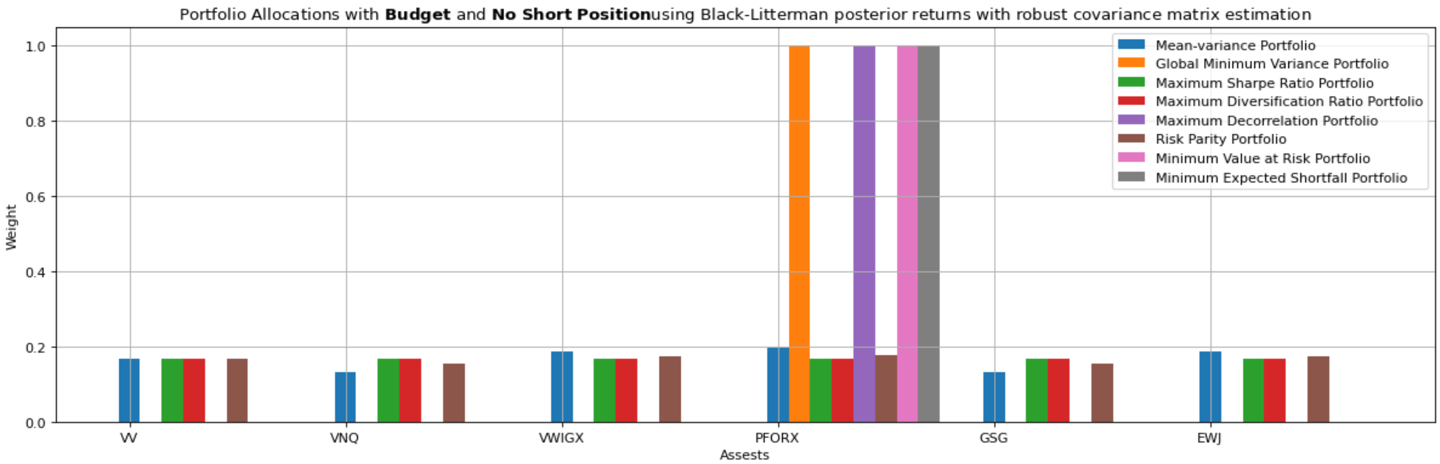
In [60]:
```
w_BL_min_ES_with_constraints = minimize_expected_shortfall_with_constraints(μ_BL, Σ, α=0.95)
w_BL_min_ES_with_constraints
```

Out[60]:
```
array([[0.00000000e+00],
       [1.44422155e-16],
       [0.00000000e+00],
       [1.00000000e+00],
       [2.14622351e-16],
       [0.00000000e+00]])
```

Note that we see the assets' weights are very concentrated in minimum value at risk portfolio and minimum expected shortfall portfolios. Again, regularization technique can be applied to further diversify assets' allocations.

In [61]:
```
fig = plt.figure(figsize=(15,5), dpi=80, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)
ax.bar(assets_tickers, w_BL_mean_var_tradeoff_with_constraints.flatten(), width=0.1, label='Mean-variance Portfolio')
ax.bar(np.arange(len(assets_tickers))+0.1, w_min_var_with_constraints.flatten(), width=0.1, label='Global Minimum Variance Portfolio')
ax.bar(np.arange(len(assets_tickers))+0.2, w_BL_max_sharpe_ratio_with_constraints.flatten(), width=0.1, label='Maximum Sharpe Ratio Portfolio
ax.bar(np.arange(len(assets_tickers))+0.3, w_BL_max_diversification_ratio_with_constraints.flatten(), width=0.1, label='Maximum Diversificati
ax.bar(np.arange(len(assets_tickers))+0.4, w_BL_max_decorrelation_with_constraints.flatten(), width=0.1, label='Maximum Decorrelation Portfol
ax.bar(np.arange(len(assets_tickers))+0.5, w_risk_parity_with_constraints.flatten(), width=0.1, label='Risk Parity Portfolio')
ax.bar(np.arange(len(assets_tickers))+0.6, w_BL_min_VaR_with_constraints.flatten(), width=0.1, label='Minimum Value at Risk Portfolio')
```

```python
ax.bar(np.arange(len(assets_tickers))+0.7, w_BL_min_ES_with_constraints.flatten(), width=0.1, label='Minimum Expected Shortfall Portfolio')
ax.set(xlabel='Assests', ylabel='Weight', title=r'Portfolio Allocations with $\bf{Budget}$ and $\bf{No}$ $\bf{Short}$ $\bf{Position}$'\
       'using Black-Litterman posterior returns with robust covariance matrix estimation')
ax.legend()
ax.grid()
```

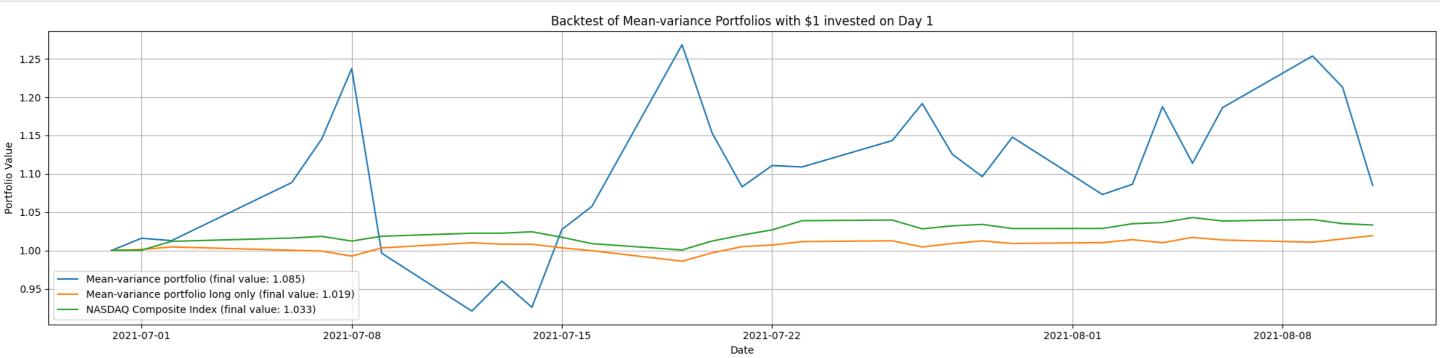


Note that the global minimum variance portfolio, maximum decorrelation portfolio, minimum value at risk portfolio, minimum expected shortfall portfolio also hit the same "corner problem" of diversification mentioned in the section of mean-variance portfolio. To further diversify these portfolios we can use the same regularizaiton techique mentioned in the section of mean-variance portfolio.

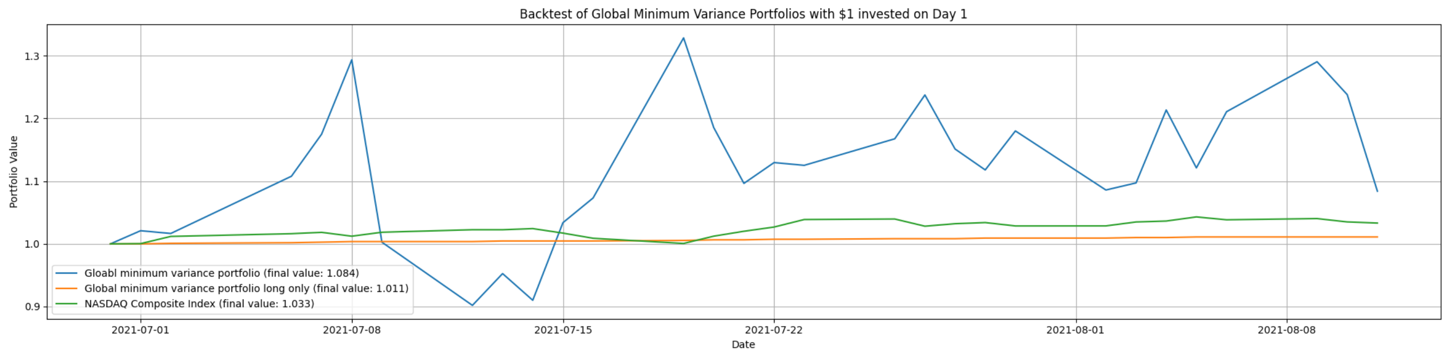
more backtestings

In this main backtesting section, we test several kinds of portfolio optimizations on the recent test trading days. We compare portfolios with and without short position for each kind of optimization.

In [62]:
```python
fig = plt.figure(figsize=(20,5), dpi=100, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)
ax.plot(returns_test @ w_BL_mean_var_tradeoff, \
        label=f'Mean-variance portfolio (final value: {round((returns_test.iloc[-1].values @ w_BL_mean_var_tradeoff)[0], 3)})')
ax.plot(returns_test @ w_BL_mean_var_tradeoff_with_constraints, \
        label=f'Mean-variance portfolio long only (final value: {round((returns_test.iloc[-1].values @ w_BL_mean_var_tradeoff_with_constraint
ax.plot(market_returns_test, \
        label=f'NASDAQ Composite Index (final value: {round(market_returns_test.iloc[-1], 3)})')
ax.set(xlabel='Date', ylabel='Portfolio Value', title='Backtest of Mean-variance Portfolios with $1 invested on Day 1')
ax.legend()
ax.grid()
```

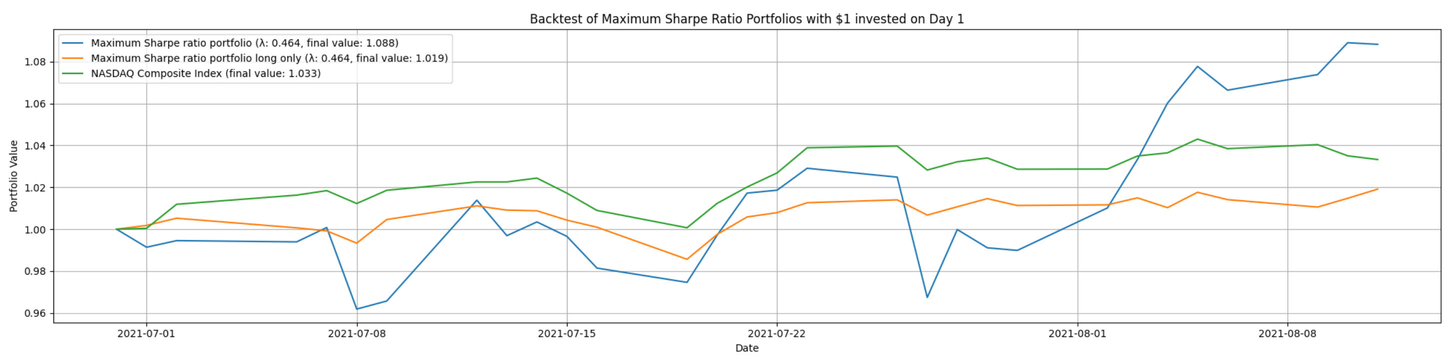


In [63]:
```python
fig = plt.figure(figsize=(20,5), dpi=100, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)
ax.plot(returns_test @ w_min_var, \
        label=f'Gloabl minimum variance portfolio (final value: {round((returns_test.iloc[-1].values @ w_min_var)[0], 3)})')
ax.plot(returns_test @ w_min_var_with_constraints, \
        label=f'Global minimum variance portfolio long only (final value: {round((returns_test.iloc[-1].values @ w_min_var_with_constraints)[
ax.plot(market_returns_test, \
        label=f'NASDAQ Composite Index (final value: {round(market_returns_test.iloc[-1], 3)})')
ax.set(xlabel='Date', ylabel='Portfolio Value', title='Backtest of Global Minimum Variance Portfolios with $1 invested on Day 1')
ax.legend()
ax.grid()
```



In [64]:

```python
fig = plt.figure(figsize=(20,5), dpi=100, tight_layout=True)
ax = fig.add_subplot(1, 1, 1)
ax.plot(returns_test @ w_BL_max_sharpe_ratio, \
        label='Maximum Sharpe ratio portfolio ' \
        f'(λ: {λ:.3f}, final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio)[0], 3)})')
ax.plot(returns_test @ w_BL_max_sharpe_ratio_with_constraints, \
        label='Maximum Sharpe ratio portfolio long only '\
        f'(λ: {λ:.3f}, final value: {round((returns_test.iloc[-1].values @ w_BL_max_sharpe_ratio_with_constraints)[0], 3)})')
ax.plot(market_returns_test, \
        label=f'NASDAQ Composite Index (final value: {round(market_returns_test.iloc[-1], 3)})')
ax.set(xlabel='Date', ylabel='Portfolio Value', title='Backtest of Maximum Sharpe Ratio Portfolios with $1 invested on Day 1')
ax.legend()
ax.grid()
```



The function below is a condensed end-to-end pipeline from portfolio selection to backtest of different optimized portfolio using excess returns computed from the Black-litterman model with robust covariance matrix estimation. Its input argument is a flag of whether the portfolio we choose are diversified by markets or not. We use this function along with the following plotting function to compare performances of different portfolio choice using different optimization methods.

In [65]:

```python
def pipeline(allocate_by_diversified_markets, allow_short_position=False):
    # choose portfolio
    if allocate_by_diversified_markets:
        assets_tickers = ['VV', 'VNQ', 'VWIGX', 'PFORX', 'GSG', 'EWJ']
    else:
        assets_tickers = ['GOOGL', 'MS', 'BX', 'ASML']
    market_ticker = 'QQQ' # to use as a strong market benchmark
    assets_prices = yf.download(assets_tickers, start='2015-01-01', end='2021-12-31', progress=False)['Adj Close'].fillna(method="ffill")
    market_prices = yf.download(market_ticker, start='2015-01-01', end='2021-12-31', progress=False)['Adj Close'].fillna(method="ffill")

    # split train/test test
    num_test_days = 30
    num_historical_days = num_test_days * 4
    returns_historical, returns_test, market_returns_test = generate_standardized_returns_data(num_test_days, num_historical_days, assets_tic

    # robust covariance matrix estimation
    direct_nonlinear_shrinkage_estimator = DirectKernel(returns_historical.values)
    Σ = direct_nonlinear_shrinkage_estimator.estimate_cov_matrix()

    w_market = generate_marketcap_weights(assets_tickers, allocate_by_diversified_markets) # market capitalization weights
    λ = 0.5/np.sqrt(w_market.T @ Σ @ w_market).flatten()[0] # risk aversion
    risk_free_rate = 0.0007 # use 1-year Treasury rate 0.07%
    τ = 1/len(returns_historical)

    # generate views
    if allocate_by_diversified_markets:
        P = np.eye(len(assets_tickers))
        Q = np.array([0.3, 0.03, 0.3, 0.005, 0.02, 0.06]).reshape(-1, 1)
    else:
        P = np.eye(len(assets_tickers))
        Q = np.array([0.5, 0.45, 0.6, 1]).reshape(-1, 1)

    # Black-litterman application
    μ_BL, Σ_BL = black_litterman(λ, Σ, w_market, τ, Q, P)

    # portfolio optimizations
    if allow_short_position == True:
        w_BL_mean_var_tradeoff = maximize_mean_variance_tradeoff_with_budget(λ, μ_BL, Σ)
```

```python
        w_min_var = minimize_variance_with_budget(Σ)
        w_BL_max_sharpe_ratio = maximize_sharpe_ratio_with_budget(risk_free_rate, μ_BL, Σ)
    else:
        w_BL_mean_var_tradeoff = maximize_mean_variance_tradeoff_with_constraints(λ, μ_BL, Σ, regularize=True)
        w_min_var = minimize_variance_with_constraints(Σ)
        w_BL_max_sharpe_ratio = maximize_sharpe_ratio_with_constraints(risk_free_rate, μ_BL, Σ)

    w_BL_max_diversification_ratio_with_constraints = maximize_diversification_ratio_with_constraints(Σ)
    w_BL_max_decorrelation_with_constraints = maximize_decorrelation_with_constraints(Σ)
    w_risk_parity_with_constraints = risk_parity_with_constraints(Σ)
    return (assets_tickers, returns_test, market_returns_test,
            w_BL_mean_var_tradeoff, w_min_var, w_BL_max_sharpe_ratio, \
            w_BL_max_diversification_ratio_with_constraints, w_BL_max_decorrelation_with_constraints, w_risk_parity_with_constraints)
```

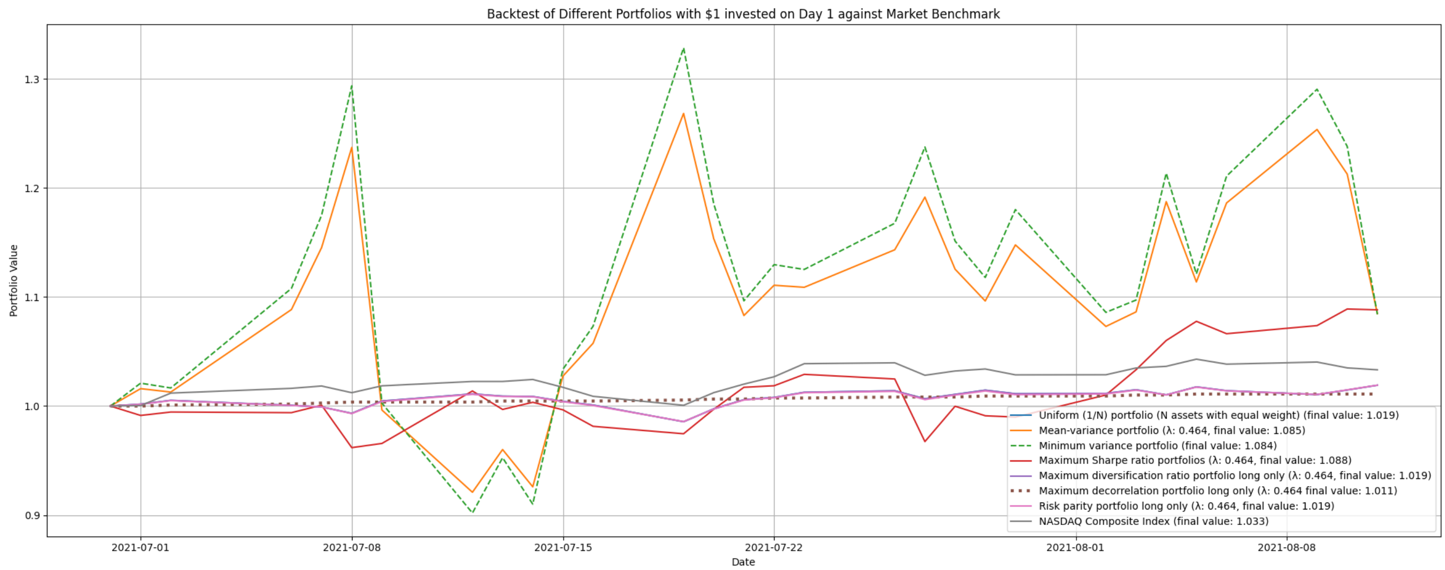
In [66]:

```python
def plot(allocate_by_diversified_markets, allow_short_position):
    (assets_tickers, returns_test, market_returns_test, \
     w_BL_mean_var_tradeoff, \
     w_min_var, \
     w_BL_max_sharpe_ratio, \
     w_BL_max_diversification_ratio_with_constraints, \
     w_BL_max_decorrelation_with_constraints, \
     w_risk_parity_with_constraints) = pipeline(allocate_by_diversified_markets, allow_short_position)

    w_equal = np.ones((len(assets_tickers), 1))/len(assets_tickers)

    fig = plt.figure(figsize=(20,8), dpi=100, tight_layout=True)
    ax = fig.add_subplot(1, 1, 1)
    ax.plot(returns_test @ w_equal, \
            label=f'Uniform (1/N) portfolio (N assets with equal weight) (final value: {round((returns_test @ w_equal).values[-1][0], 3)})')
    ax.plot(returns_test @ w_BL_mean_var_tradeoff, \
            label=f'Mean-variance portfolio (λ: {λ:.3f}, final value: {round((returns_test @ w_BL_mean_var_tradeoff).values[-1][0], 3)})')
    ax.plot(returns_test @ w_min_var, \
            linestyle='dashed',
            label=f'Minimum variance portfolio (final value: {round((returns_test @ w_min_var).values[-1][0], 3)})')
    ax.plot(returns_test @ w_BL_max_sharpe_ratio, \
            label=f'Maximum Sharpe ratio portfolios ' \
            f'(λ: {λ:.3f}, final value: {round((returns_test @ w_BL_max_sharpe_ratio).values[-1][0], 3)})')
    ax.plot(returns_test @ w_BL_max_diversification_ratio_with_constraints, \
            label='Maximum diversification ratio portfolio long only ' \
            f'(λ: {λ:.3f}, final value: {round((returns_test @ w_BL_max_diversification_ratio_with_constraints).values[-1][0], 3)})')
    ax.plot(returns_test @ w_BL_max_decorrelation_with_constraints, \
            linestyle='dotted',
            linewidth=3,
            label=f'Maximum decorrelation portfolio long only ' \
            f'(λ: {λ:.3f} final value: {round((returns_test @ w_BL_max_decorrelation_with_constraints).values[-1][0], 3)})')
    ax.plot(returns_test @ w_risk_parity_with_constraints, \
            label=f'Risk parity portfolio long only ' \
            f'(λ: {λ:.3f}, final value: {round((returns_test @ w_risk_parity_with_constraints).values[-1][0], 3)})')
    ax.plot(market_returns_test, \
            label=f'NASDAQ Composite Index (final value: {round(market_returns_test.iloc[-1], 3)})')
    ax.set(xlabel='Date', ylabel='Portfolio Value', title='Backtest of Different Portfolios with $1 invested on Day 1 against Market Benchmar
    ax.legend()
    ax.grid()
```

In [67]:

```python
# portfolio of diversified assets
plot(allocate_by_diversified_markets=True, allow_short_position=True)
```

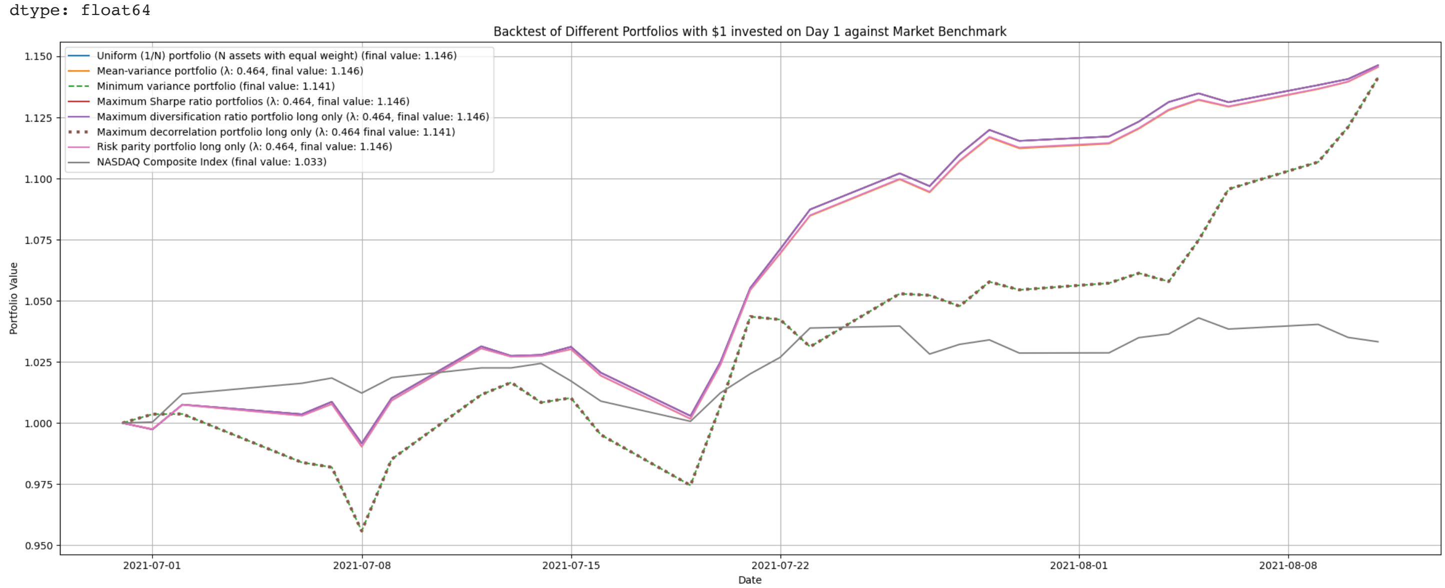
```
assets' market capitalization weights:
VV       0.176187
VNQ      0.361935
VWIGX    0.340081
PFORX    0.061172
GSG      0.006396
EWJ      0.054231
dtype: float64
```



In [68]:

```
# portfolio of hand-picked (and concentrated) assets with high "active risk"
plot(allocate_by_diversified_markets=False, allow_short_position=False)
```

```
assets' market capitalization weights:
GOOGL    0.753983
MS       0.078283
BX       0.033817
ASML     0.133918
dtype: float64
```



From the above 2 plots, we have 1 portfolio with diversified ETF shares from different markets, while the budget is contrained, short positions are allowed. We see mean-variance (with regularization) long only and global minimum variance long only portfolios outperform the market benchmark as well as the other kinds portfolios. It is also very interesting to observe that the maximum Sharpe ratio portfolio long only doesn't perform very well albeit having positive returns in the backtest period.

On the other hand, the second portfolio with high active risks also outperforms the market, though not having as high absolute return as the first portfolio. This finding supports the popularity of fundamental research as investors often times end up allocating into riskier assets they have strong belief in. We also see the returns of maximum decorrelation portfolio and minimum variance portfolio overlap with each other, as their assets' allocation weights are identical given the few number of assets selected in the portfolio. In addition, although maximum diversification portfolio and risk parity portfolio don't have much difference in assets' allocation weights, these optimization approaches do have pretty obvious effect on actively chosen riskier portfolio.

Acknowledgement

The delegate would like to thank Dr. Richard Diamond for delivering the workshops on the topic of portfolio construction, and his patiencce answering many questions through emails after lectures. In addition, the delegate would like to thank all CQF faculty and staff members for organizing these high-quality course materials not just on the topic of portfolio construction, but on many other ones beyond. It's been a great time learning experience with you all!

References:

1. https://scikit-learn.org/stable/modules/covariance.html
2. Rousseeuw, Peter J. "Least median of squares regression." Journal of the American statistical association 79.388 (1984): 871-880.
3. Ledoit, Olivier, and Michael Wolf. "A well-conditioned estimator for large-dimensional covariance matrices." Journal of multivariate analysis 88.2 (2004): 365-411.
4. Ledoit, Olivier, and Michael Wolf. Direct nonlinear shrinkage estimation of large-dimensional covariance matrices. No. 264. Working Paper, 2017.
5. CQF Module 2 Lecture 2 slides

6. Idzorek, Thomas. "A step-by-step guide to the Black-Litterman model: Incorporating user-specified confidence levels." Forecasting expected returns in the financial markets. Academic Press, 2007. 17-38.
7. Meucci, Attilio. "The black-litterman approach: Original model and extensions." Shorter version in, THE ENCYCLOPEDIA OF QUANTITATIVE FINANCE, Wiley (2010).
8. He, Guangliang, and Robert Litterman. "The intuition behind Black-Litterman model portfolios." Available at SSRN 334304 (2002).
9. Corvalán, Alejandro. "Well diversified efficient portfolios." Documentos de Trabajo (Banco Central de Chile) 336 (2005): 1.
10. Marek Capiński and Tomasz Zastawniak. Mathematics for Finance: An Introduction to Financial Engineering. 2nd ed. Springer, 2011. Chap. 3. ISBN: 978-0-85729-081-6.
11. CQF Risk Budgeting slides
12. Roncalli, Thierry. Introduction to risk parity and budgeting. CRC Press, 2013.
13. Lopez de Prado, Marcos. "Machine Learning Asset Allocation (Presentation Slides)." Available at SSRN 3469964 (2019).