Unison Industry Project - Portfolio Optimization

Pradeepta Das, Jill Shah, Apeksha Jain, Brodie Gay (supervisor)

May 2021

1 Introduction

Large institutional investors own a variety of assets spanning corporate equities and debt, mortgages, commercial real estate and treasuries. We want to improve these institutional investment portfolios by providing them access to homeowner equity.

The objective of this project is to create a portfolio optimization tool which takes a sample institutional portfolio weighting and shows how diversifying into homeowner equity investments can improve risk/return of the portfolio.

The tool should be able to take a current portfolio holding of a predetermined set of investment asset classes, show expected return and risk (plus other relevant metrics such as Sharpe or VAR), and display them on an efficient frontier. Then, it should be able to show how a small re-allocation to Unison investments can improve those metrics. Finally it should also propose a reasonable allocation to the unison homeowner equity.

The project can be split into four different milestones.

- Determining a set of investment asset classes that large institutions typically invest in. Determine expected return, volatility and correlation for those asset classes using historical sample estimates.
- Estimate the expected return, volatility and correlations of the unison investments with the other asset classes. Using our estimate of Alpha and Beta of Unison vs. Case Shiller, and the above estimates, determine the return, volatility and correlation vector for Unison investments.
- Write an optimizer with proper constraints
- Build an application to show the allocation, optimization results and efficient frontier.

2 Asset Class Determination and Data Collection

In order to determine a realistic allocation of portfolios of the institutional investors, we analysed various endowment funds such as Yale, Harvard and UC; soverign funds such as GIC, ARIA, Korea Investment Fund; Pension funds such as Japan Pension fund and CalPERS. We also looked at the portfolios of JPMorgan asset management and Blackrock. From this analysis we concluded that we can broadly categorize the investment into 4 categories.

- Equities

U.S. large cap equities	MSCI USA Index
Emerging large cap equities	MSCI Emerging Markets Index
U.S. small cap equities	MSCI USA Small Cap Index
Global ex-U.S. large cap equities	MSCI World ex-US Index

- Fixed Income

Bloomberg Barclays US Agg Total Return Index	LBUSTRUU Index
Bloomberg Barclays US Credit Treasure	LUCRTRUU Index
Bloomberg Barclays US Treasury	LUATTRUU Index
Bloomberg Barclays MBS Convent	BC2YTRUU Index
Bloomberg Barclays U.S. Government Long TR Index	LGL1TRUU Index
JPMorgan Monthly EMBIs	JPEIDIVR Index
Bloomberg Barclays US Corporate	LF98TRUU Index
JPMorgan GBI-EM GI	GBIEMCOR Index
Bloomberg Barclays US Govt Inflation	BCIT1T Index
Bloomberg Barclays Global Aggregate	BRTUTRUU Index
US Treasury 3M Bill MM Yield	USBMMY3M Index

- Private market

HFR Asset Wghted Comp	HFRIAWC Index
Private Equity Total Return Index	PRIVEXD Index

- Homeowner equity (Core Real Estate, REIT index, Case Shiller index)

National Association of Real Estate Investment Trusts	NAREIT Index
Case Shiller index	CSUSHPINSA

3 Data Analysis

To understand the impact that time horizon has on returns and volatility, we analysed the annualised returns and annualised volatility for different time horizons - 1 month, 3 months, 1 year and 3 years for all the indices. The results have been summarised below:

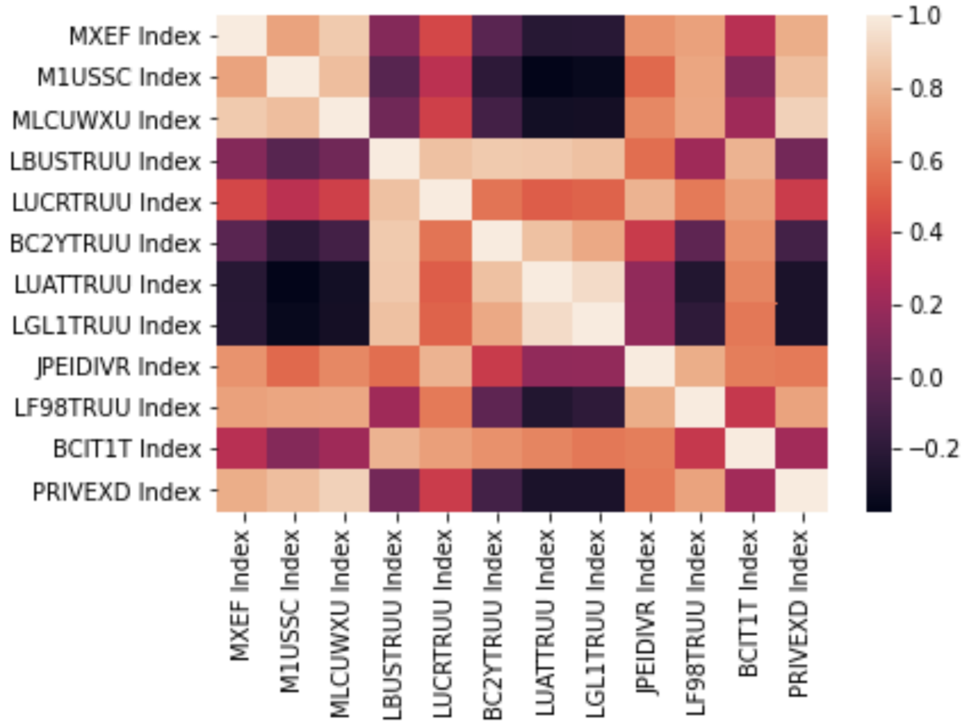
	Ret1MAnnual	Ret3MAnnual	Ret1YAnnual	Ret3YAnnual
M1US Index	9.44%	9.48%	9.01%	8.23%
MXEF Index	3.73%	3.70%	3.06%	3.28%
MLCUWXU Index	2.57%	2.50%	2.09%	2.11%
LBUSTRUU Index	5.28%	5.34%	5.42%	5.14%
LUCRTRUU Index	6.07%	6.14%	6.16%	5.84%
LUATTRUU Index	4.92%	4.99%	5.14%	4.83%
LGL1TRUU Index	7.17%	7.39%	7.76%	7.15%
JPEIDIVR Index	9.13%	9.40%	9.45%	9.10%
LF98TRUU Index	7.16%	7.18%	7.01%	6.86%
CSUSHPINSA	4.10%	4.09%	4.04%	4.04%
NAREIT	9.10%	9.14%	9.17%	9.31%

	Vol1MAnnual	Vol3MAnnual	Vol1YAnnual	Vol3YAnnual
M1US Index	15.21%	15.52%	17.05%	18.78%
MXEF Index	22.84%	25.59%	26.00%	20.99%
MLCUWXU Index	16.44%	17.59%	18.70%	16.53%
LBUSTRUU Index	3.48%	3.58%	3.76%	3.80%
LUCRTRUU Index	5.19%	5.30%	5.26%	4.36%
LUATTRUU Index	4.32%	4.44%	4.29%	4.25%
LGL1TRUU Index	10.13%	10.04%	8.80%	5.49%
JPEIDIVR Index	11.51%	11.54%	10.28%	8.44%
LF98TRUU Index	8.70%	9.96%	10.18%	8.40%
CSUSHPINSA	2.42%	3.95%	5.79%	8.88%
NAREIT	18.96%	18.95%	19.93%	18.07%

As can be seen from the results above, the returns and volatility vary across time periods. Hence, in order to include all the seasonality and cyclical shocks and to be consistent with our portfolio optimisation results, we consider a time horizon of 1 year from hereCas on.

4 Index Selection

To select the indices, we looked at the correlations between them. The heat maps for 1M and 1Y maturity have been given below.



After analysing the heat maps, MLCUWXU Index, LUCRTRUU Index, BC2YTRUU Index, LGL1TRUU Index, BCIT1T Index and PRIVEXD Index were dropped as they exhibited high correlation for both 1M and 1Y with one of the other indices.

So the final set of indicies that are used in the portfolio optimization are M1US Index, MXEF Index, LBUSTRUU Index, LUATTRUU Index, JPEIDIVR Index, LF98TRUU Index, and NAREIT as the final set of indices in our portfolio.

We also use Case Shiller as a proxy to generate the time series for Unison's homeowner equity using the following equation:

$$Unison = 2.1 * (0.01 + CaseShiller) \tag{1}$$

and include this time series in our portfolio. However, for our analysis below, we are using Case-Shiller as our proxy index for Unison.

5 Mean-Variance Portfolio Optimisation

5.1 Without Constraints - Derivation:

Let U be the utility function with the Lagrangian incorporated for the constraint $h^T\mathbf{1} = 1$ where h represent the weights for the assets in the portfolio. Hence U, which is the utility function that needs to be maximised, can be mathematically represented as:

$$max_h U = h^T\mu - \frac{\gamma}{2}h^T\Sigma h - \lambda * (h^T\mathbf{1} - 1) \tag{2}$$

Equating the first derivative of U with respect to h to 0, we get:

$$\frac{\partial U}{\partial h} = \mu - \gamma\Sigma h - \lambda * \mathbf{1} = 0 \tag{3}$$

$$h = \Sigma^{-1}\Big(\frac{\mu - \lambda\mathbf{1}}{\gamma}\Big) \tag{4}$$

$$\frac{\partial U}{\partial \lambda} = h^T\mathbf{1} - 1 = 0 \tag{5}$$

Substituting the value of h from equation (3) in equation (4) we get:

$$\Big(\frac{\mu^T - \lambda\mathbf{1}^T}{\gamma}\Big)\Sigma^{-1}\mathbf{1} - 1 = 0 \tag{6}$$

$$\lambda = \frac{\mu^T\Sigma^{-1}\mathbf{1} - \gamma}{\mathbf{1}^T\Sigma^{-1}\mathbf{1}} \tag{7}$$

Substituting the value of λ in h in equation (3) we get:

$$h = \frac{\Sigma^{-1}}{\gamma}\left[\mu - \frac{\mu^T\Sigma^{-1}\mathbf{1}}{\mathbf{1}^T\Sigma^{-1}\mathbf{1}}\mathbf{1}\right] + \frac{\Sigma^{-1}\mathbf{1}}{\mathbf{1}^T\Sigma^{-1}\mathbf{1}} \tag{8}$$

5.1.1 Results:

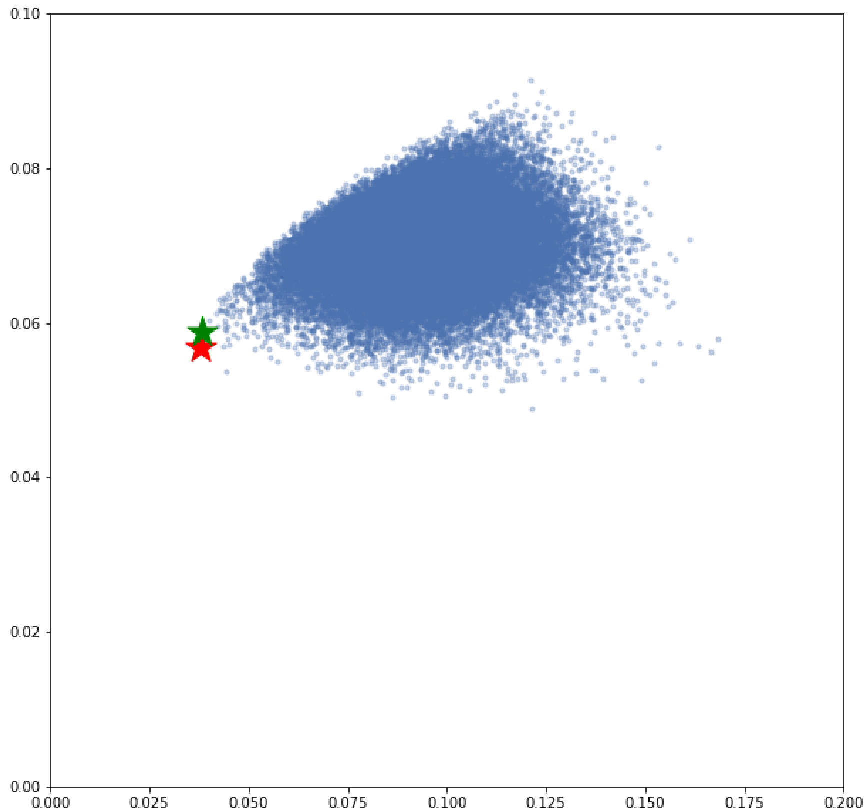
The results for the unconstrained mean-variance optimisation have been summarised below.

Index Name	Weights
M1US Index	450.57%
MXEF Index	1462.92%
LBUSTRUU Index	19917.10%
LUATTRUU Index	6391.68%
JPEIDIVR Index	-522.78%
LF98TRUU Index	2029.25%
NAREIT	-59009.09%
Unison	-2571.36%
Unison	34309.09%

5.2 With Constraints

5.2.1 MonteCarlo

In MonteCarlo simulation we started with assigning random weights to our assets, keeping the sum of weights equal to 1. We ran a simulation creating 10,000 portfolios which allowed us to generate an efficient frontier. Below is the image of Montecarlo results

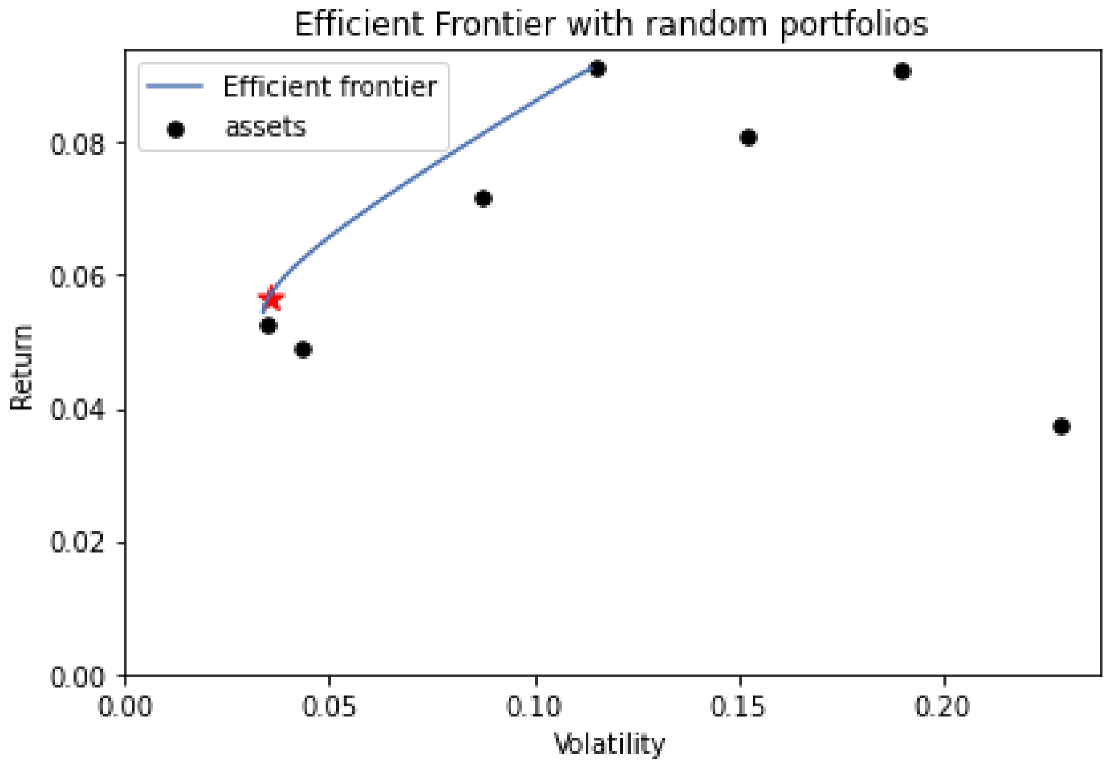


The results have been summarised below:

	Max Sharpe Portfolio	Min Variance Portfolio
Returns	5.88%	5.68%
Volatility	3.82%	3.77%
Sharpe	1.015	0.976

5.2.2 Pypfportfolio

Montecarlo simulation though robust, is computationally very expensive. Pypfportfolio offers a better way to solve the mean variance optimization problem. Below we have run the optimization using this library on the same set of assets and under the same constraints:



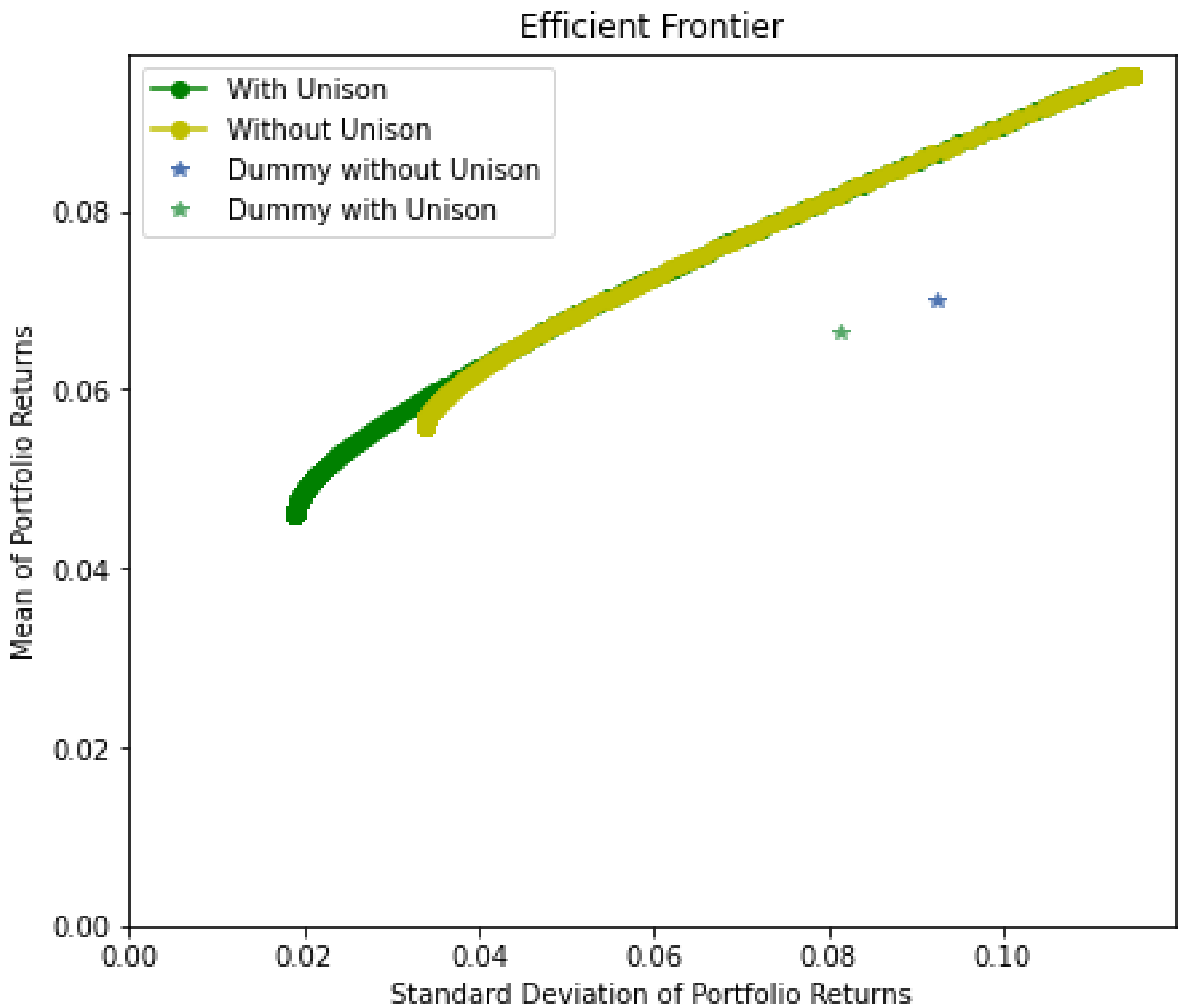
The results have been summarised below:

	Max Sharpe Portfolio
Returns	5.6%
Volatility	3.1%
Sharpe	1.16

As expected, results are clearly better than MonteCarlo simulation.

6 Adding Unison Home Equity to the portfolio

As we saw earlier, convex optimization packages have clear advantages over Monte Carlo simulation. To be able to incorporate all the constraints we intend to use in this project we have decided to go ahead with CVXOPT package. To show the benefit of adding Unison Home Equity to the client portfolio we have created a dummy portfolio, consisting equal weights of all the selected assets. To this we add 10% Unison home equity and proportionately reducing weight from other assets.



7 Web Application

We have used the stream-lit package in python to build the web application which can be used to showcase our results to potential investors.

- The objective of the application is to be able to take in user defined the current portfolio weights, the constraints for the weights to be used in the optimization problem and also the unison beta and alpha parameters relative to the Case-Shiller index.
- The application is also hosted using Heroku platform and can be access via the URL. https://portfolio-optimization-unison.herokuapp.com/

Below we show the two views for the web application that was built.

Figure 1: Web Application - View 1

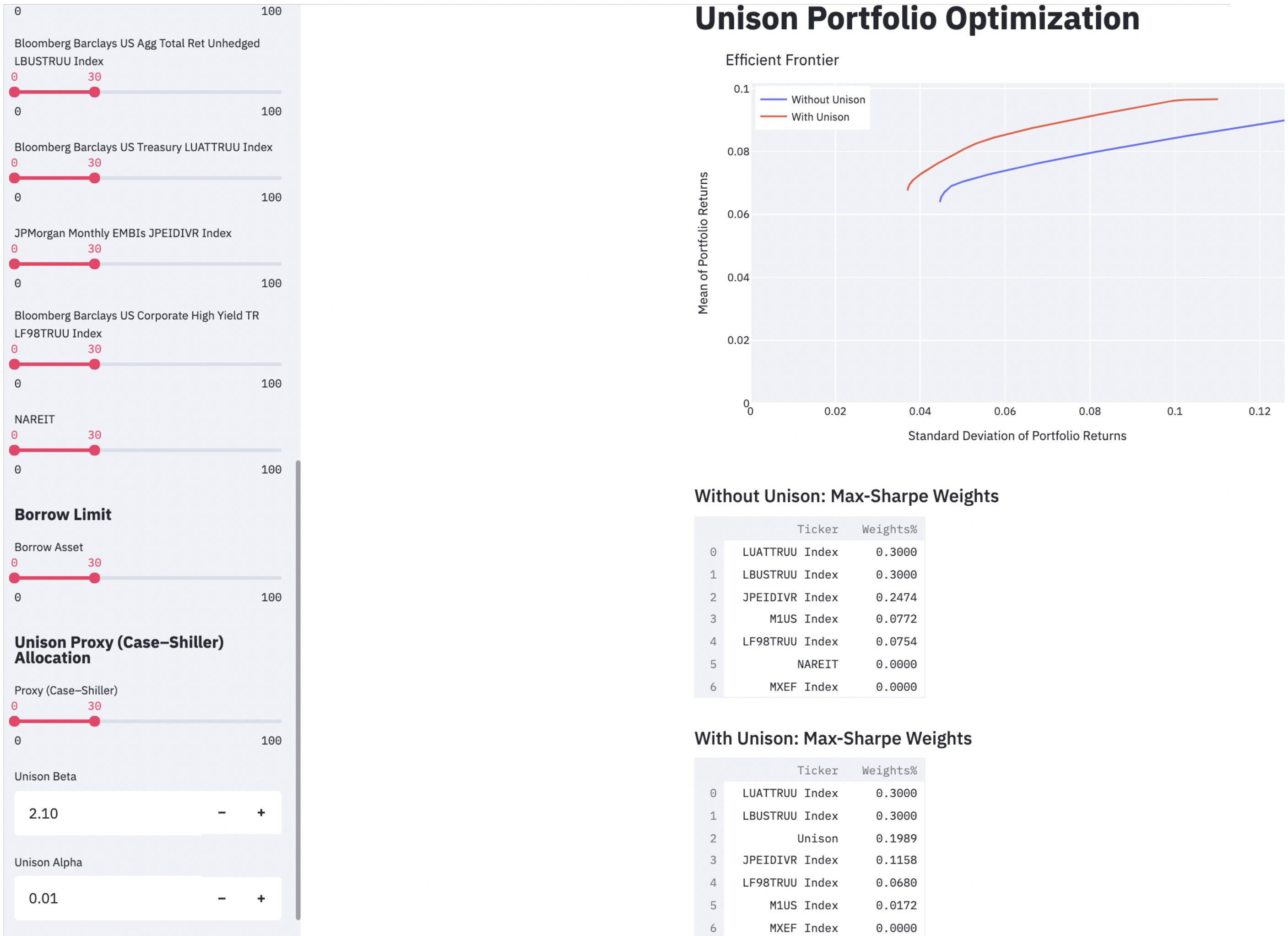


Figure 2: Web Application - View 2

8 Codes

8.1 Code for Optimizer

:

```
import numpy as np
import pandas as pd
import cvxopt as opt
import streamlit as st
from cvxopt import blas, solvers
import matplotlib.pyplot as plt
import json
pd.options.display.float_format = '{:.2%}'.format

class UOptimizer:
    def __init__(self, path, include_unison=True, beta=2.1, alpha=0.01):
        """ does what it says """
        self.path = path
        self.data = self.load_data(path)
        self.include_unison = include_unison
        self.beta = beta
        self.alpha = alpha
        self.weights = None
        self.period = 12

        self.return_to_use = None
        self.annualized_return_to_use = None
        self.annualized_vol_to_use = None

        #set returns and volatility
        self.set_returns_vols()

    @staticmethod
    def get_name_dict():
        return {'M1US Index':"MSCI USA Net TR USD",
                'MXEF Index':"MSCI Emerging Markets",
                'LBUSTRUU Index':"Bloomberg Barclays US Agg Total Ret Unhedged",
                'LUATTRUU Index':"Bloomberg Barclays US Treasury",
                'JPEIDIVR Index':"JPMorgan Monthly EMBIs",
                'LF98TRUU Index':"Bloomberg Barclays US Corporate High Yield TR",
                'CSUSHPINSA': "Case-Shiller Index"
                }

    def set_unison_alpha_beta(self, alpha, beta):
        """ does what it says """
```

```
        self.alpha = alpha
        self.beta = beta
        self.set_returns_vols()

    def set_returns_vols(self):
        """ does what it says """
        self.return_to_use = self.logReturns(self.data, self.period).dropna()
        if self.include_unison:
            self.return_to_use["Unison"] = self.beta * (self.return_to_use['CSUSHPINSA']
        self.annualized_vol_to_use = self.return_to_use.std() * np.sqrt(12/self.period)
        self.annualized_return_to_use = self.return_to_use * (12/self.period)
        self.cleanup()

    @staticmethod
    def load_data(path):
        """ does what it says """
        data = pd.read_excel(path, sheet_name='Sheet1').iloc[2:]
        data.columns = data.iloc[0]
        data = data.iloc[1:].reset_index().drop(columns = ['index'])
        data.set_index('Date', inplace=True)

        #keep the indices which have atleast 60% data!
        data.dropna(thresh=len(data)*0.9, axis=1, inplace=True)
        data = data.astype('float')

        #interpolate the missing values, sort using index, take the log!
        data = data.interpolate()
        data = data.sort_index()
        data = np.log(data)
        return data

    @staticmethod
    def logReturns(df, period):
        """ does what it says """
        logRet = df.diff(period)
        return logRet.dropna()

    def cleanup(self):
        """ does what it says """
        stocks_to_be_dropped = ['LUCRTRUU Index','LGL1TRUU Index', 'MLCUWXU Index', 'CSU
        self.return_to_use = self.return_to_use.drop(stocks_to_be_dropped, axis=1)
        self.annualized_vol_to_use = self.annualized_vol_to_use.drop(stocks_to_be_droppe
        self.annualized_return_to_use = self.annualized_return_to_use.drop(stocks_to_be_

    @staticmethod
```

```python
def optimal_portfolio(returns, low_weight_bound, high_weight_bound):
    """ does what it says """
    # Turn off progress printing
    solvers.options['show_progress'] = False
    returns = np.asmatrix(returns.T)                    # -> (n_assets, n_observations)
    n_assets = len(returns)

    # Vector of desired returns
    N = n_assets*(int(1e+2))
    mus = [10 ** (5.0 * t / N - 1.0) for t in range(N)]

    # Obtain expected returns and covariance
    m1 = np.mean(returns, axis=1)                           # Mean returns
    c1 = np.cov(returns, bias=True)                         # Volatility (in terms of s
    # Convert to cvxopt matrices
    pbar = opt.matrix(m1)
    S = opt.matrix(c1)

    # Create constraint matrices
    G = opt.matrix(np.vstack((-np.eye(n_assets), np.eye(n_assets))))
    h = opt.matrix(np.vstack((low_weight_bound, high_weight_bound)))
    A = opt.matrix(1.0, (1, n_assets))
    b = opt.matrix(1.0)
    # Calculate efficient frontier weights using quadratic programming
    portfolios = [solvers.qp(mu * S, -pbar, G, h, A, b)['x'] for mu in mus]
    sol = solvers.qp(S, -pbar, G, h, A, b)

    ## CALCULATE RISKS AND RETURNS FOR FRONTIER
    weights = [np.asarray(x) for x in portfolios]
    returns = np.asarray([blas.dot(pbar, x) for x in portfolios])
    risks = np.asarray([np.sqrt(blas.dot(x, S * x)) for x in portfolios])
    sharpe = returns/risks
    max_sharpe_idx = np.argmax(sharpe)
    min_vol_idx = np.argmin(risks)

    #UOptimizer.matplot_eff_frontier(returns, risks, sharpe)
    return weights, np.asarray(returns), np.asarray(risks), sharpe

@staticmethod
def matplot_eff_frontier(returns, risks, sharpe):
    """matplot lib version of efficient frontier"""
    ax_sharpe_idx = np.argmax(sharpe)
    min_vol_idx = np.argmin(risks)

    max_sharpe_idx = np.argmax(sharpe)
```

```
        min_vol_idx = np.argmin(risks)

        # Plot Efficient Frontier
        fig, ax = plt.subplots()
        plt.plot(risks, returns, 'y-o')
        plt.plot(risks[max_sharpe_idx], returns[max_sharpe_idx], '*', label = 'max_sharp
        plt.plot(risks[min_vol_idx], returns[min_vol_idx], '*', label = 'min_vol')
        plt.title('Efficient Frontier')
        plt.ylabel('Mean of Portfolio Returns')
        plt.xlabel('Standard Deviation of Portfolio Returns')
        plt.grid()
        plt.legend()
        st.pyplot(fig)

    def parse_weights(self, weights_dict):
        """ does what it says """
        low_weight_bound = [i[0]/100 for k, i in weights_dict.items() if self.include_un
        high_weight_bound = [i[1]/100 for k, i in weights_dict.items() if self.include_u
        return np.asarray(low_weight_bound).reshape(-1,1), np.asarray(high_weight_bound)

    def summarize(self, returns, risks, sharpe):
        ret = self.return_to_use
        weights = self.weights
        ind_opt = np.argmax(sharpe)            # Index of selected portfolio

        opt_portfolio = {}
        opt_portfolio['return'] = returns[ind_opt]
        opt_portfolio['risk'] = risks[ind_opt]
        opt_portfolio['sharpe'] = sharpe[ind_opt]

        wt = weights[ind_opt]/sum(weights[ind_opt])
        ind_w = np.flip(np.argsort(wt, axis=0), axis=0)
        opt_portfolio['weights'] = wt[ind_w]
        ind_w = ind_w.ravel().tolist()
        sym1 = pd.DataFrame(list(ret))

        sym=sym1.loc[ind_w]

        #sym = [str(sym[k][0][0]) for k in range(len(sym))]
        opt_portfolio['stocks'] = sym

        output = pd.DataFrame(columns=["Ticker","Weights%"])
        output["Ticker"] = sym[0]
        output["Weights%"] = wt[ind_w]
        output = output.reset_index(drop=True)
```

```
        st.write(output)

    def optimize_main(self, weights_dict):
        """ does what it says """
        ret = self.return_to_use
        low_weight_bound, high_weight_bound = self.parse_weights(weights_dict)
        weights, returns, risks, sharpe = self.optimal_portfolio(ret,
                                                                  low_weight_bound,
                                                                  high_weight_bound)

        self.weights = weights
        return returns, risks, sharpe
```

8.2 Code for Web Application

:

```
    % options to customize output of pythoncode
    % see section 5.3 Available options starting at page 16
import streamlit
import os
import glob
import time
import multiprocessing
import logging

import streamlit as st
import numpy as np
import pandas as pd
pd.options.display.float_format = '{:.2%}'.format

import plotly.graph_objects as go
from optimizer import UOptimizer
#from Inputs_Parallel import get_possible_scenarios
import chart_studio.plotly as py

import matplotlib.pyplot as plt

def plotly_eff_frontier(optimizer, optimizer_unison, weights):
    """ does what it says """
    # Graphing Function #####
    returns, risks, sharpe = optimizer.optimize_main(weights)
    fig = go.Figure(data=[go.Scatter(x=risks,
                            y=returns,
                            #hoveron=sharpe,
```

```
                        mode= "lines", #'lines+markers', #"lines"
                        name = 'Without Unison',
                        #line=go.scatter.Line(color="gray"),
                        #showlegend=False)
                        marker=dict(
                                size=10,
                                color=sharpe, #set color equal to a variable
                                colorscale='Viridis', # one of plotly colorscales
                                showscale=True
                            )
                        )
                     ])

returns1, risks1, sharpe1 = optimizer_unison.optimize_main(weights)
fig.add_trace(
    go.Scatter(
        x=risks1,
        y=returns1,
        mode="lines",#"markers",
        name="With Unison",
        marker=dict(
                    size=3,
                    color=sharpe, #set color equal to a variable
                    #colorscale='Viridis', # one of plotly colorscales
                    showscale=True
                )
        #line=dict(color="black")
    )
)

fig.update_layout(legend=dict(
    yanchor="top",
    y=0.99,
    xanchor="left",
    x=0.01
))

fig['layout']['yaxis'].update(autorange=True, rangemode='tozero')
fig['layout']['xaxis'].update(autorange=True, rangemode='tozero')
fig.update_layout(hovermode='x unified')
fig.update_layout(title='Efficient Frontier', autosize=True,
                  xaxis=dict(
                            title=dict(
                              text='Standard Deviation of Portfolio Returns'
                        )),
```

```
                      yaxis=dict(
                                title=dict(
                                  text='Mean of Portfolio Returns'
                            )),
                      #width=800, height=800,
                      margin=dict(l=40, r=40, b=40, t=40))
    st.plotly_chart(fig)

    st.subheader('Without Unison: Max-Sharpe Weights')
    #optimizer.matplot_eff_frontier(x, y, sharpe)
    optimizer.summarize(returns, risks, sharpe)

    st.subheader('With Unison: Max-Sharpe Weights')
    #optimizer_unison.matplot_eff_frontier(x1, y1, sharpe1)
    optimizer_unison.summarize(returns, risks, sharpe1)


def main(optimizer, optimizer_unison):
    # Side Bar ########################################################
    #think about using 'collapsible_container' in fuutre

    st.sidebar.subheader('Current Portfolio Weights')
    pf_wt = {}
    for c in optimizer.annualized_return_to_use.columns:
        if c != "CSUSHPINSA":
            pf_wt[c] = st.sidebar.number_input(label=optimizer.get_name_dict().get(c, ""
                                                   value=0.1)

    st.sidebar.subheader('Asset Weight Limits For Optimization')
    weights = {}

    for c in optimizer.annualized_return_to_use.columns:
        if c != "CSUSHPINSA": # don't use case-shiller here
            weights[c] = st.sidebar.slider(label=optimizer.get_name_dict().get(c, "") +
                                              min_value=0,
                                              max_value=100,
                                              step=1,
                                              value=(0, 30))

    st.sidebar.subheader('Borrow Limit')
    a = st.sidebar.slider(label="Borrow Asset",
                                              min_value=0,
                                              max_value=100,
                                              step=1,
                                              value=(0, 30))
```

```
    st.sidebar.subheader('Unison Proxy (Case{Shiller) Allocation')
    weights['Unison'] = st.sidebar.slider(label="Proxy (Case{Shiller)",
                                          min_value=0,
                                          max_value=100,
                                          step=1,
                                          value=(0, 30))

    #run_button = st.sidebar.button(label='Run Optimization')

    #in case we want to give user the flexibility to
    unison_beta = st.sidebar.number_input("Unison Beta", value=2.1)
    unison_alpha = st.sidebar.number_input("Unison Alpha", value=0.01)
    optimizer_unison.set_unison_alpha_beta(unison_alpha, unison_beta)

    # App ###################################################
    st.title("Unison Portfolio Optimization")
    plotly_eff_frontier(optimizer, optimizer_unison, weights)


#@st.cache
def load_optimizer():
    optimizer = UOptimizer('timeseriesUpdated.xlsx', include_unison=False)
    optimizer_unison = UOptimizer('timeseriesUpdated.xlsx')
    return (optimizer, optimizer_unison)

if __name__ == '__main__':
    logging.basicConfig(level=logging.CRITICAL)
    optimizer, optimizer_unison = load_optimizer()
    main(optimizer, optimizer_unison)
```

UnisonPortOpt

May 13, 2021

1 Unison Project

Pradeepta Das, Jill Shah, Apeksha Jain

[20]:
```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image

pd.options.display.float_format = '{:.2%}'.format
```

2 Data Preprocessing

[21]:
```
data = pd.read_excel('timeseriesUpdated.xlsx', sheet_name='Sheet1').iloc[2:]
data.columns = data.iloc[0]
data = data.iloc[1:].reset_index().drop(columns = ['index'])
data.set_index('Date', inplace=True)
```

[22]:
```
pd.options.display.float_format = '{:.2f}'.format
data.head()
```

[22]:
```
2           M1US Index MXEF Index M1USSC Index MLCUWXU Index LBUSTRUU Index  \
Date
2021-03-31   11085.83    1316.43       727.88        952.58         2311.35
2021-02-26   10687.87    1339.26       713.16         932.7          2340.58
2021-01-29   10420.14    1329.57       669.87        907.54         2374.87
2020-12-31   10520.81    1291.26       645.58        919.38         2392.02
2020-11-30   10108.22    1205.07       598.89        881.32         2388.73

2           LUCRTRUU Index BC2YTRUU Index LUATTRUU Index LGL1TRUU Index  \
Date
2021-03-31        3331.52         255.07        2450.55         4251.2
2021-02-26        3385.44         256.13        2488.92        4472.86
2021-01-29        3445.22         258.32        2534.92        4733.83
2020-12-31        3486.71         257.91         2559.4         4908.6
2020-11-30        3470.86         257.14        2565.34        4966.07
```

```
2          JPEIDIVR Index  … GBIEMCOR Index BCIT1T Index BRTUTRUU Index  \
Date                       …
2021-03-31         949.99  …         141.76       351.88         130.77
2021-02-26         959.23  …         146.25       352.82         134.69
2021-01-29         984.36  …         150.13       359.43         138.36
2020-12-31         995.16  …         151.72       358.42         140.29
2020-11-30         976.62  …         146.53        354.4          138.0

2          USBMMY3M Index MXUS0INF Index HFRIAWC Index PRIVEXD Index  \
Date
2021-03-31            NaN         892.68            NaN       1973.14
2021-02-26            NaN         820.13            NaN       1865.77
2021-01-29            NaN         851.75            NaN       1809.54
2020-12-31            NaN         867.28        1519.21       1824.41
2020-11-30           0.09         867.34        1469.65       1738.74

2          M1USIRE Index CSUSHPINSA  NAREIT
Date
2021-03-31       1463.44        NaN 8962.61
2021-02-26       1400.81     238.82 8500.61
2021-01-29       1350.31     236.33 8248.44
2020-12-31       1348.44     236.31 8261.85
2020-11-30       1305.62     234.45 8040.25

[5 rows x 21 columns]
```

[23]:
```
pd.options.display.float_format = '{:.2%}'.format
```

[24]:
```
isNACount = (data.isna().sum())
isNACount
```

[24]:
```
2
M1US Index         0
MXEF Index         0
M1USSC Index      80
MLCUWXU Index      0
LBUSTRUU Index     1
LUCRTRUU Index     2
BC2YTRUU Index    74
LUATTRUU Index     1
LGL1TRUU Index     3
JPEIDIVR Index     7
LF98TRUU Index     1
GBIEMCOR Index   163
BCIT1T Index      33
BRTUTRUU Index   163
```

```
USBMMY3M Index    283
MXUSOINF Index    206
HFRIAWC Index     211
PRIVEXD Index     115
M1USIRE Index     246
CSUSHPINSA          1
NAREIT              0
dtype: int64
```

[25]:
```
#keep the indices which have atleast 90% data!
data.dropna(thresh=len(data)*0.9, axis=1, inplace=True)
data = data.astype('float')
data.isna().sum()
print("Out of", len(isNACount), "chosen indices, only", len(data.columns),␣
 ↪"remaining ",
      "after filtering out the series which have a lot of missing data.")
```

```
Out of 21 chosen indices, only 11 remaining  after filtering out the series
which have a lot of missing data.
```

[26]:
```
#interpolate the missing values, sort using index, take the log!
data = data.interpolate()
data = data.sort_index()
data = np.log(data)
```

[27]:
```
def calcReturns(df, period):
    dfRet = df.diff(period)
    return dfRet.dropna()

def logReturns(df, period):
    logRet = df.diff(period)
    return logRet.dropna()
```

[28]:
```
ret1MLog = logReturns(data, 1).dropna()
ret3MLog = logReturns(data, 3).dropna()
ret1YLog = logReturns(data, 12).dropna()
ret3YLog = logReturns(data, 36).dropna()

annualisedVol1MLog = ret1MLog.std() * np.sqrt(12)
annualisedVol3MLog = ret3MLog.std() * np.sqrt(4)
annualisedVol1YLog = ret1YLog.std() * np.sqrt(1)
annualisedVol3YLog = ret3YLog.std() * np.sqrt(1/3)

ret1MAnnulaisedLog = ret1MLog*12
ret3MAnnulaisedLog = ret3MLog*4
ret1YAnnulaisedLog = ret1YLog
ret3YAnnulaisedLog = ret3YLog*(1/3)
```

3 Average Log Annualized Return and Log Annualized Vol

[29]:
```
AnnualisedRet = pd.DataFrame()
AnnualisedRet['Ret1MAnnual'] = np.mean(ret1MAnnulaisedLog)
AnnualisedRet['Ret3MAnnual'] = np.mean(ret3MAnnulaisedLog)
AnnualisedRet['Ret1YAnnual'] = np.mean(ret1YAnnulaisedLog)
AnnualisedRet['Ret3YAnnual'] = np.mean(ret3YAnnulaisedLog)
AnnualisedRet.index = list(annualisedVol1MLog.index)
#AnnualisedRet.to_latex()

AnnualisedRet
```

[29]:

	Ret1MAnnual	Ret3MAnnual	Ret1YAnnual	Ret3YAnnual
M1US Index	9.44%	9.48%	9.01%	8.23%
MXEF Index	3.73%	3.70%	3.06%	3.28%
MLCUWXU Index	2.57%	2.50%	2.09%	2.11%
LBUSTRUU Index	5.28%	5.34%	5.42%	5.14%
LUCRTRUU Index	6.07%	6.14%	6.16%	5.84%
LUATTRUU Index	4.92%	4.99%	5.14%	4.83%
LGL1TRUU Index	7.17%	7.39%	7.76%	7.15%
JPEIDIVR Index	9.13%	9.40%	9.45%	9.10%
LF98TRUU Index	7.16%	7.18%	7.01%	6.86%
CSUSHPINSA	4.10%	4.09%	4.04%	4.04%
NAREIT	9.10%	9.14%	9.17%	9.31%

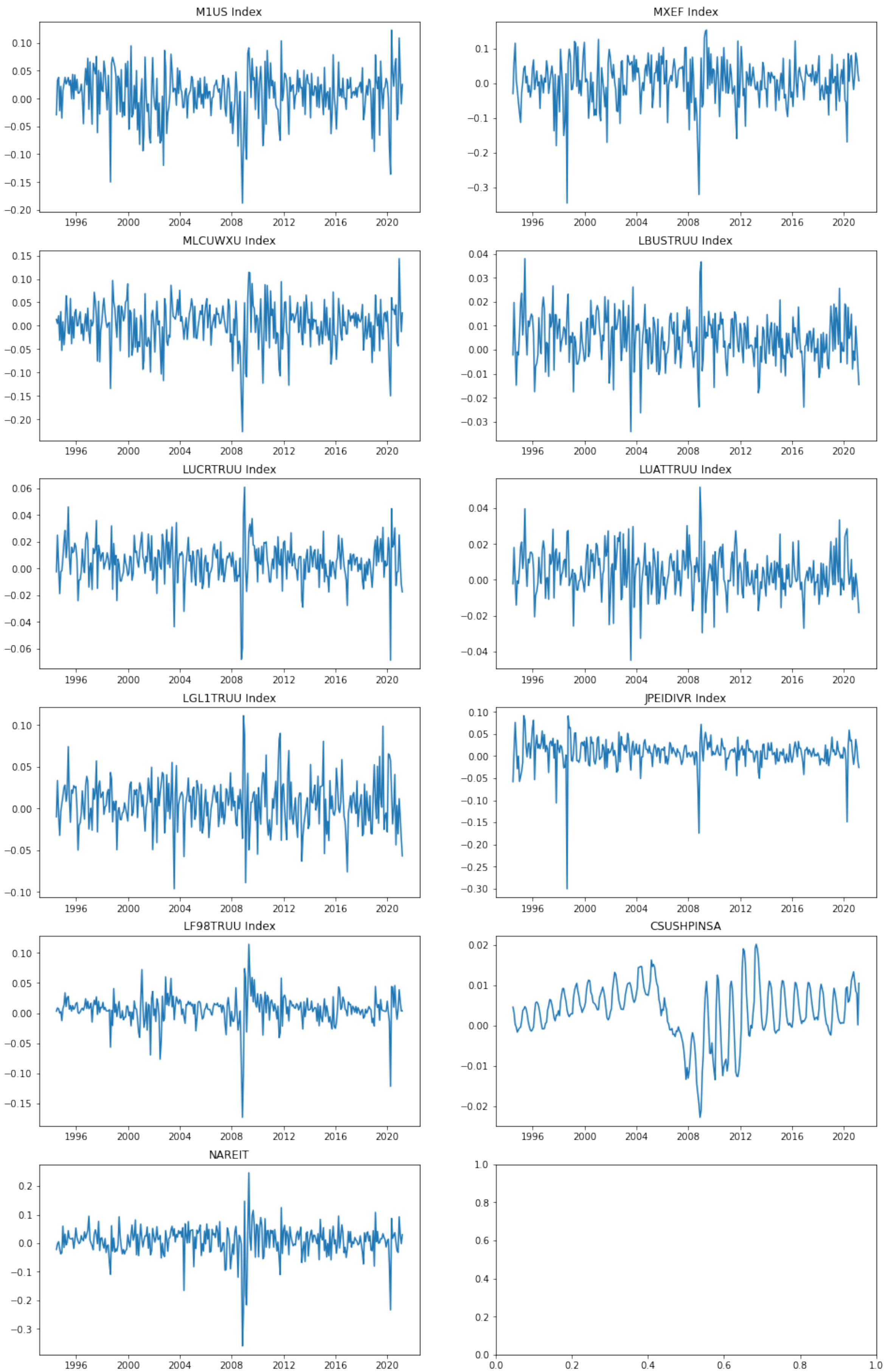
[30]:
```
AnnualisedVol = pd.DataFrame()
AnnualisedVol['Vol1MAnnual'] = annualisedVol1MLog
AnnualisedVol['Vol3MAnnual'] = annualisedVol3MLog
AnnualisedVol['Vol1YAnnual'] = annualisedVol1YLog
AnnualisedVol['Vol3YAnnual'] = annualisedVol3YLog
AnnualisedVol.index = list(annualisedVol3YLog.index)
#AnnualisedVol.to_latex()

AnnualisedVol
```

[30]:

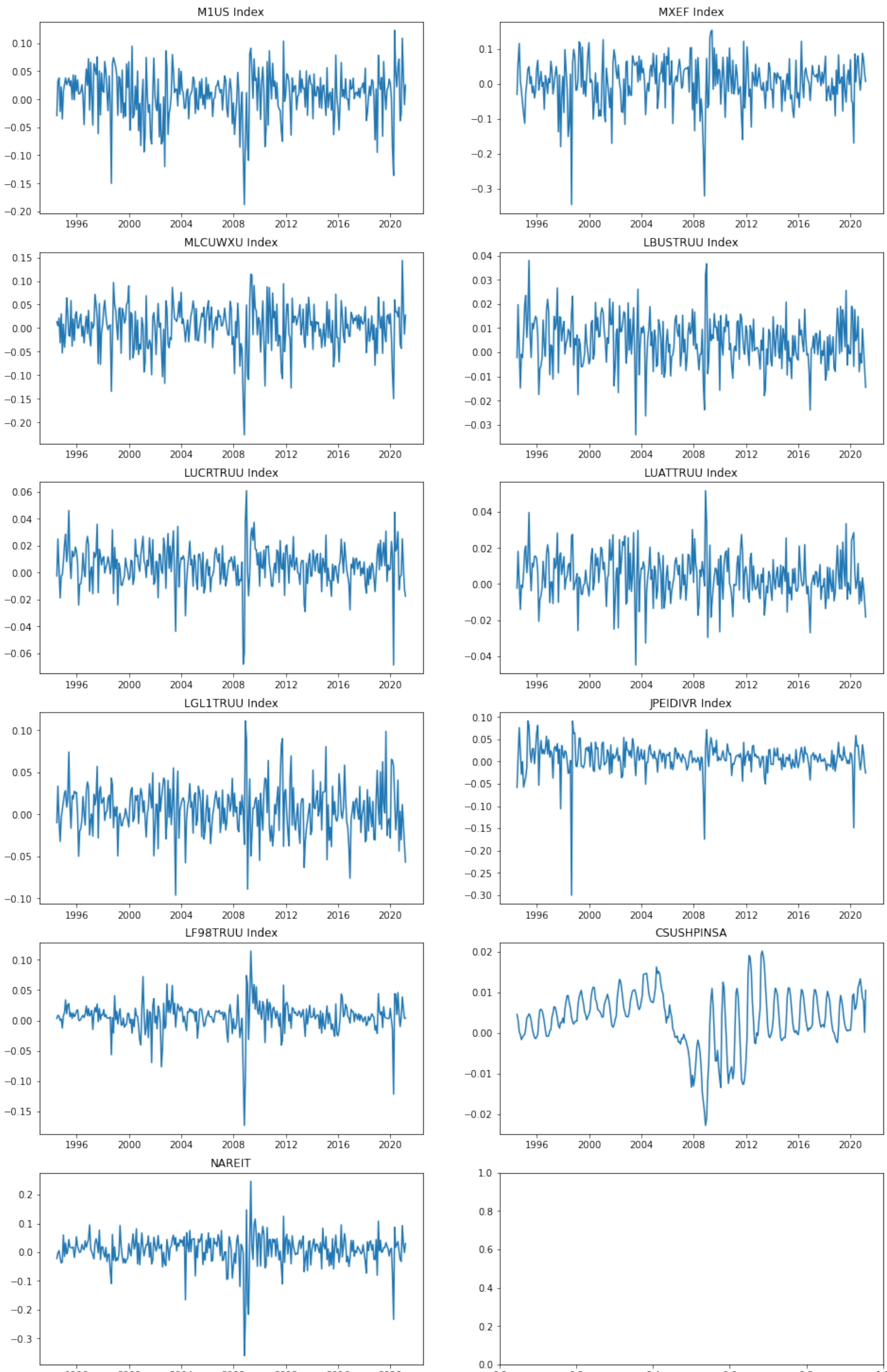
	Vol1MAnnual	Vol3MAnnual	Vol1YAnnual	Vol3YAnnual
M1US Index	15.21%	15.52%	17.05%	18.78%
MXEF Index	22.84%	25.59%	26.00%	20.99%
MLCUWXU Index	16.44%	17.59%	18.70%	16.53%
LBUSTRUU Index	3.48%	3.58%	3.76%	3.80%
LUCRTRUU Index	5.19%	5.30%	5.26%	4.36%
LUATTRUU Index	4.32%	4.44%	4.29%	4.25%
LGL1TRUU Index	10.13%	10.04%	8.80%	5.49%
JPEIDIVR Index	11.51%	11.54%	10.28%	8.44%
LF98TRUU Index	8.70%	9.96%	10.18%	8.40%
CSUSHPINSA	2.42%	3.95%	5.79%	8.88%
NAREIT	18.96%	18.95%	19.93%	18.07%

4 Log return plots

[31]:
```
fig, axes = plt.subplots(6, 2, figsize=(16,2))
axs = axes.ravel()
fig.subplots_adjust(top=10)
for i in range(0,len(ret1MLog.columns)):
    axs[i].plot(ret1MLog.iloc[:,i])
    axs[i].set_title(ret1MLog.columns[i])
```

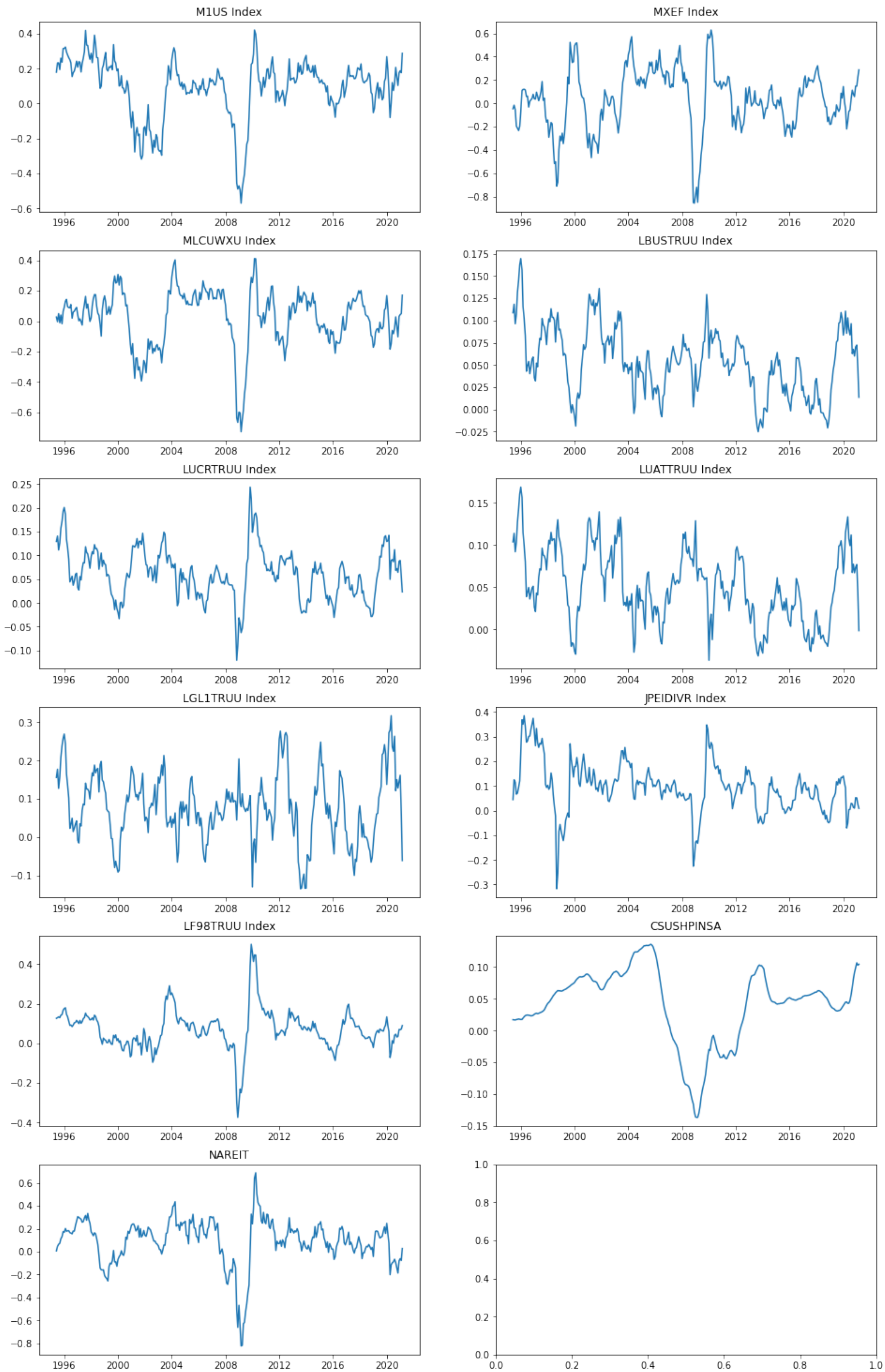

M1US Index
MXEF Index
MLCUWXU Index
LBUSTRUU Index
LUCRTRUU Index
LUATTRUU Index
LGL1TRUU Index
JPEIDIVR Index
LF98TRUU Index
CSUSHPINSA
NAREIT
1996
2000
2004
2008
2012
2016
2020

[32]:
```
fig, axes = plt.subplots(6, 2, figsize=(16,2))
axs = axes.ravel()
fig.subplots_adjust(top=10)
for i in range(0,len(ret3MLog.columns)):
    axs[i].plot(ret1MLog.iloc[:,i])
    axs[i].set_title(ret1MLog.columns[i])
```


M1US Index
MXEF Index
MLCUWXU Index
LBUSTRUU Index
LUCRTRUU Index
LUATTRUU Index
LGL1TRUU Index
JPEIDIVR Index
LF98TRUU Index
CSUSHPINSA
NAREIT

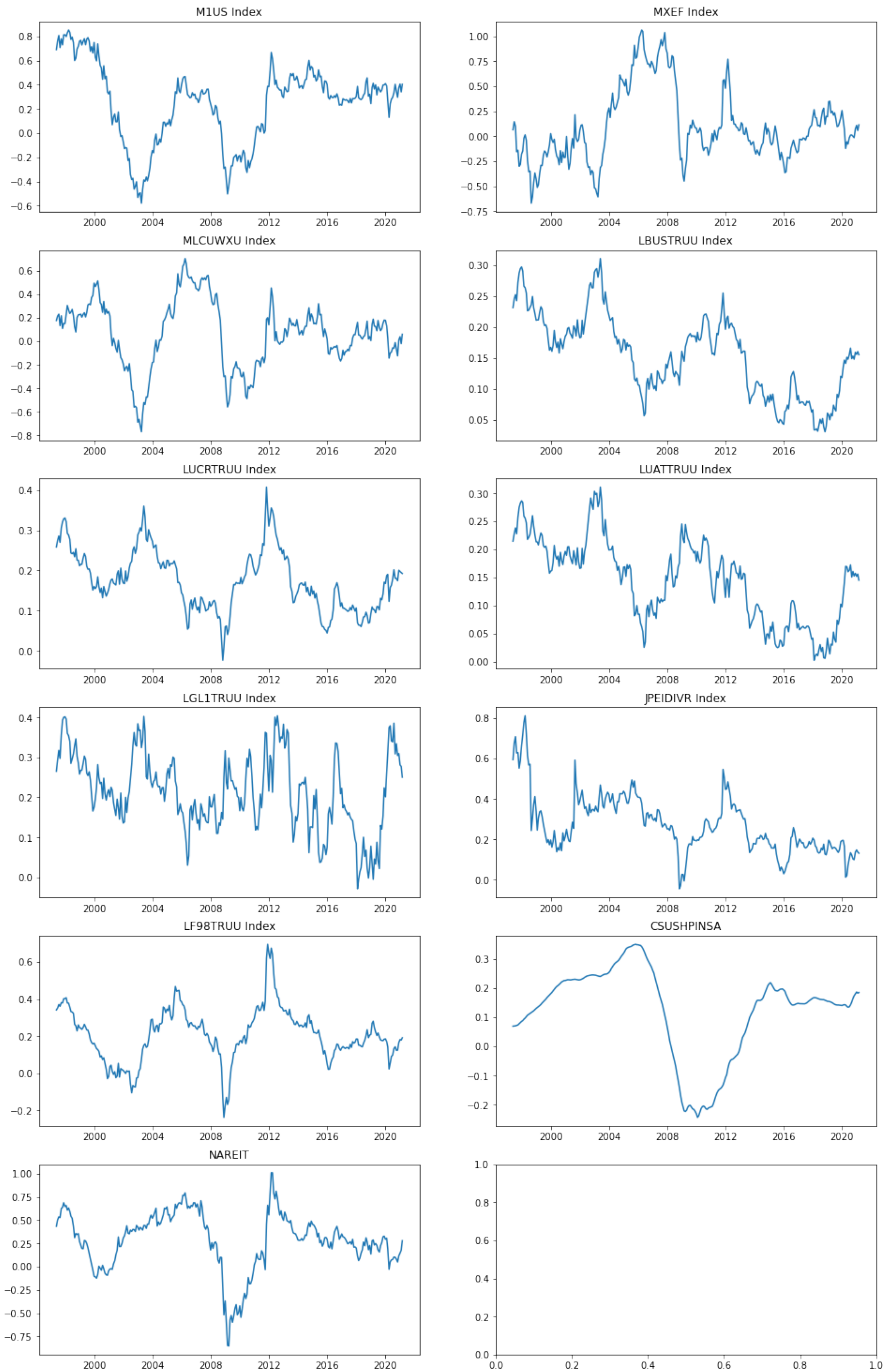
[33]:

```
fig, axes = plt.subplots(6, 2, figsize=(16,2))
axs = axes.ravel()
fig.subplots_adjust(top=10)
for i in range(0,len(ret1YLog.columns)):
    axs[i].plot(ret1YLog.iloc[:,i])
    axs[i].set_title(ret1YLog.columns[i])
```


M1US Index
MXEF Index
MLCUWXU Index
LBUSTRUU Index
LUCRTRUU Index
LUATTRUU Index
LGL1TRUU Index
JPEIDIVR Index
LF98TRUU Index
CSUSHPINSA
NAREIT
1996
2000
2004
2008
2012
2016
2020

[34]:

```
fig, axes = plt.subplots(6, 2, figsize=(16,2))
axs = axes.ravel()
fig.subplots_adjust(top=10)
for i in range(0,len(ret3YLog.columns)):
    axs[i].plot(ret3YLog.iloc[:,i])
    axs[i].set_title(ret3YLog.columns[i])
```


M1US Index
MXEF Index
MLCUWXU Index
LBUSTRUU Index
LUCRTRUU Index
LUATTRUU Index
LGL1TRUU Index
JPEIDIVR Index
LF98TRUU Index
CSUSHPINSA
NAREIT
2000
2004
2008
2012
2016
2020
0.0
0.2
0.4
0.6
0.8
1.0

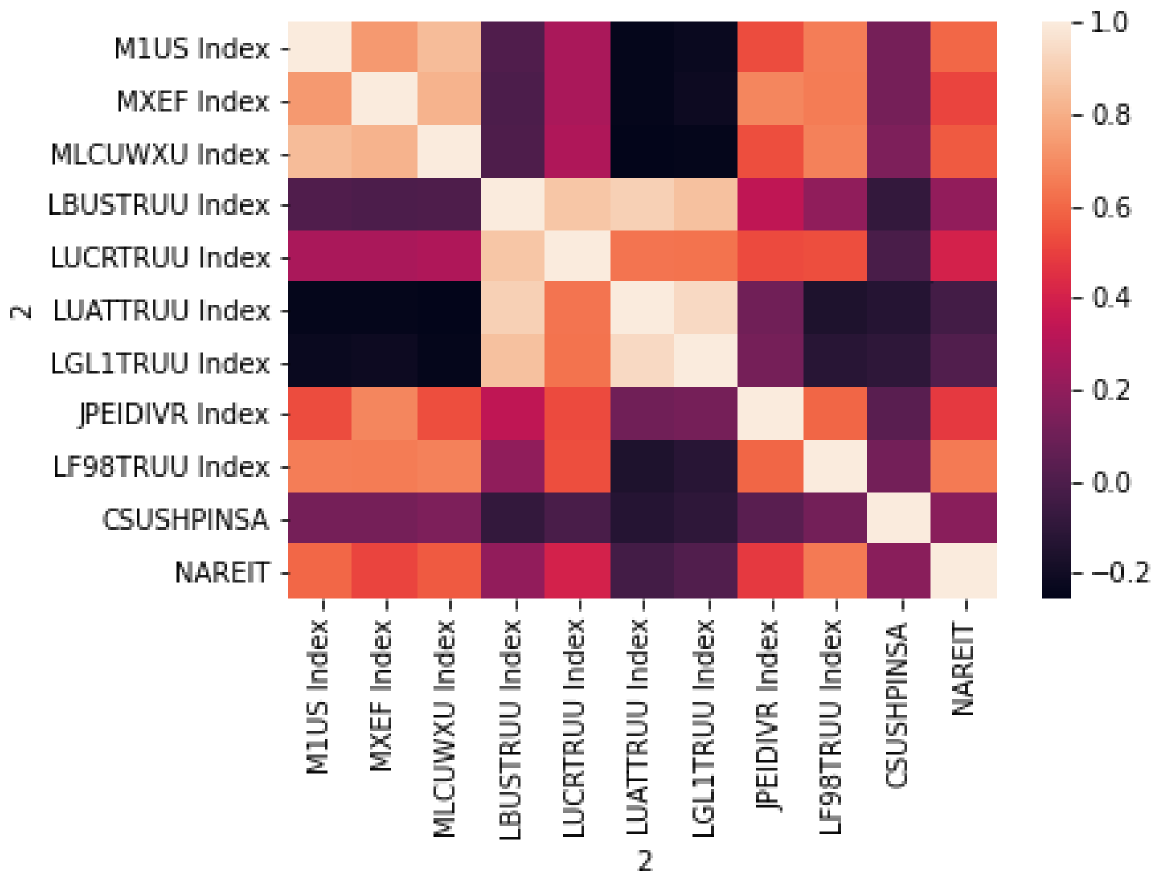
5 Correlation

[35]:
```
ret1MLog.corr()
```

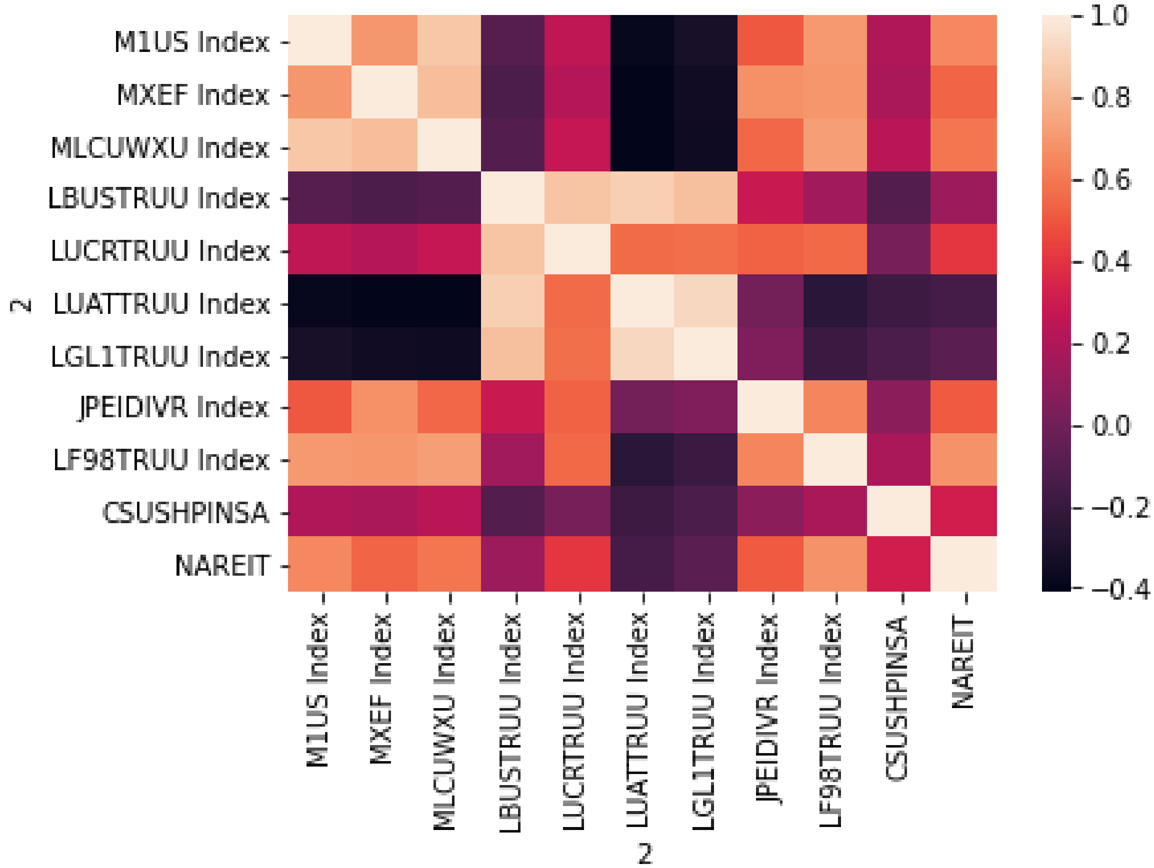
[35]:
```
2               M1US Index  MXEF Index  MLCUWXU Index  LBUSTRUU Index  \
2
M1US Index         100.00%      73.58%         84.48%           0.89%
MXEF Index          73.58%     100.00%         81.50%          -0.06%
MLCUWXU Index       84.48%      81.50%        100.00%           0.44%
LBUSTRUU Index       0.89%      -0.06%          0.44%         100.00%
LUCRTRUU Index      27.47%      27.21%         28.92%          87.50%
LUATTRUU Index     -24.10%     -24.23%        -25.27%          91.04%
LGL1TRUU Index     -21.87%     -21.22%        -24.02%          86.10%
JPEIDIVR Index      53.33%      68.58%         53.90%          33.59%
LF98TRUU Index      65.78%      65.71%         66.98%          19.99%
CSUSHPINSA          12.09%      12.29%         14.49%          -8.27%
NAREIT              60.16%      51.43%         56.73%          20.69%

2               LUCRTRUU Index  LUATTRUU Index  LGL1TRUU Index  \
2
M1US Index              27.47%         -24.10%         -21.87%
MXEF Index              27.21%         -24.23%         -21.22%
MLCUWXU Index           28.92%         -25.27%         -24.02%
LBUSTRUU Index          87.50%          91.04%          86.10%
LUCRTRUU Index         100.00%          63.47%          62.99%
LUATTRUU Index          63.47%         100.00%          93.69%
LGL1TRUU Index          62.99%          93.69%         100.00%
JPEIDIVR Index          52.54%          10.77%          12.03%
LF98TRUU Index          53.80%         -15.20%         -11.69%
CSUSHPINSA              -1.00%         -13.01%          -9.64%
NAREIT                  40.70%          -3.12%           1.26%

2               JPEIDIVR Index  LF98TRUU Index  CSUSHPINSA  NAREIT
2
M1US Index              53.33%          65.78%      12.09%  60.16%
MXEF Index              68.58%          65.71%      12.29%  51.43%
MLCUWXU Index           53.90%          66.98%      14.49%  56.73%
LBUSTRUU Index          33.59%          19.99%      -8.27%  20.69%
LUCRTRUU Index          52.54%          53.80%      -1.00%  40.70%
LUATTRUU Index          10.77%         -15.20%     -13.01%  -3.12%
LGL1TRUU Index          12.03%         -11.69%      -9.64%   1.26%
JPEIDIVR Index         100.00%          59.43%       3.72%  48.43%
LF98TRUU Index          59.43%         100.00%      11.76%  64.94%
CSUSHPINSA               3.72%          11.76%     100.00%  17.97%
```

```
NAREIT                        48.43%            64.94%        17.97% 100.00%
```

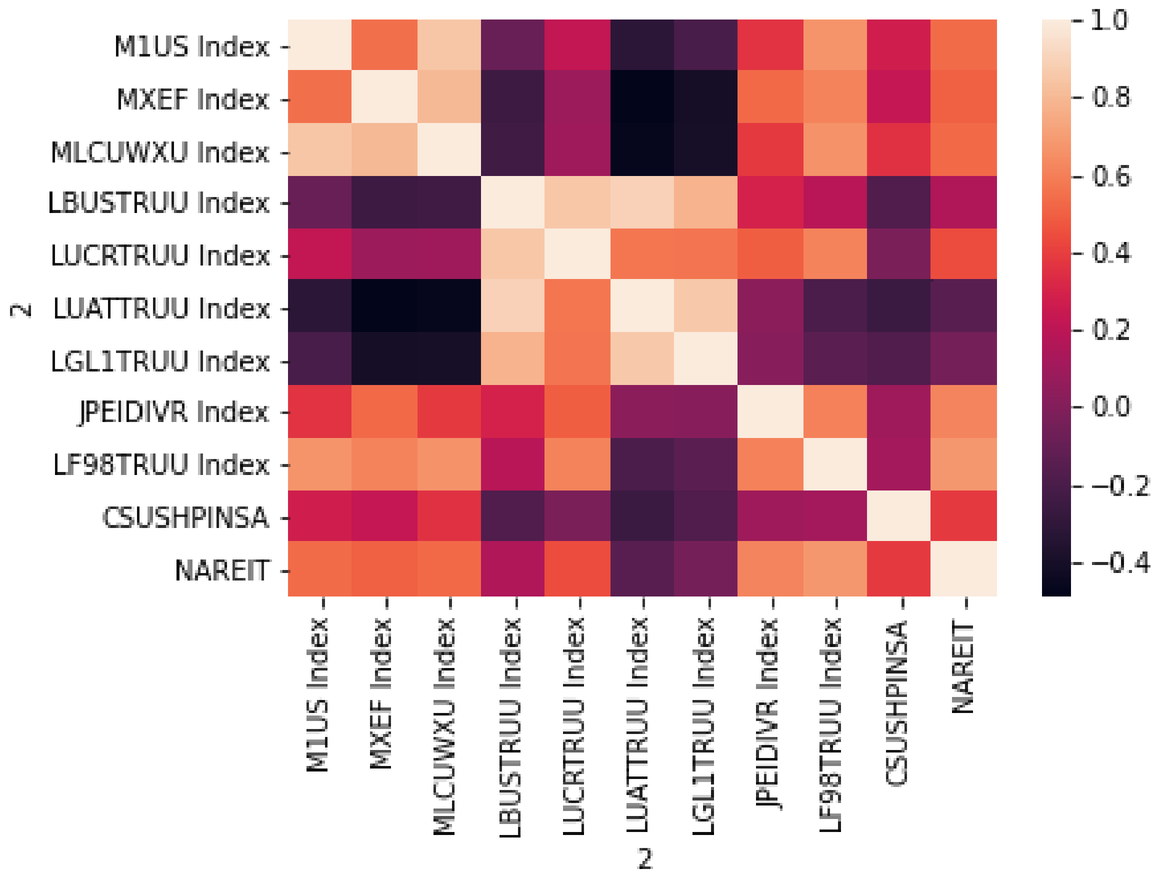
[36]:
```python
heatMap1 = sns.heatmap(ret1MLog.corr())
#plt.savefig("heatmap1M.png")
```



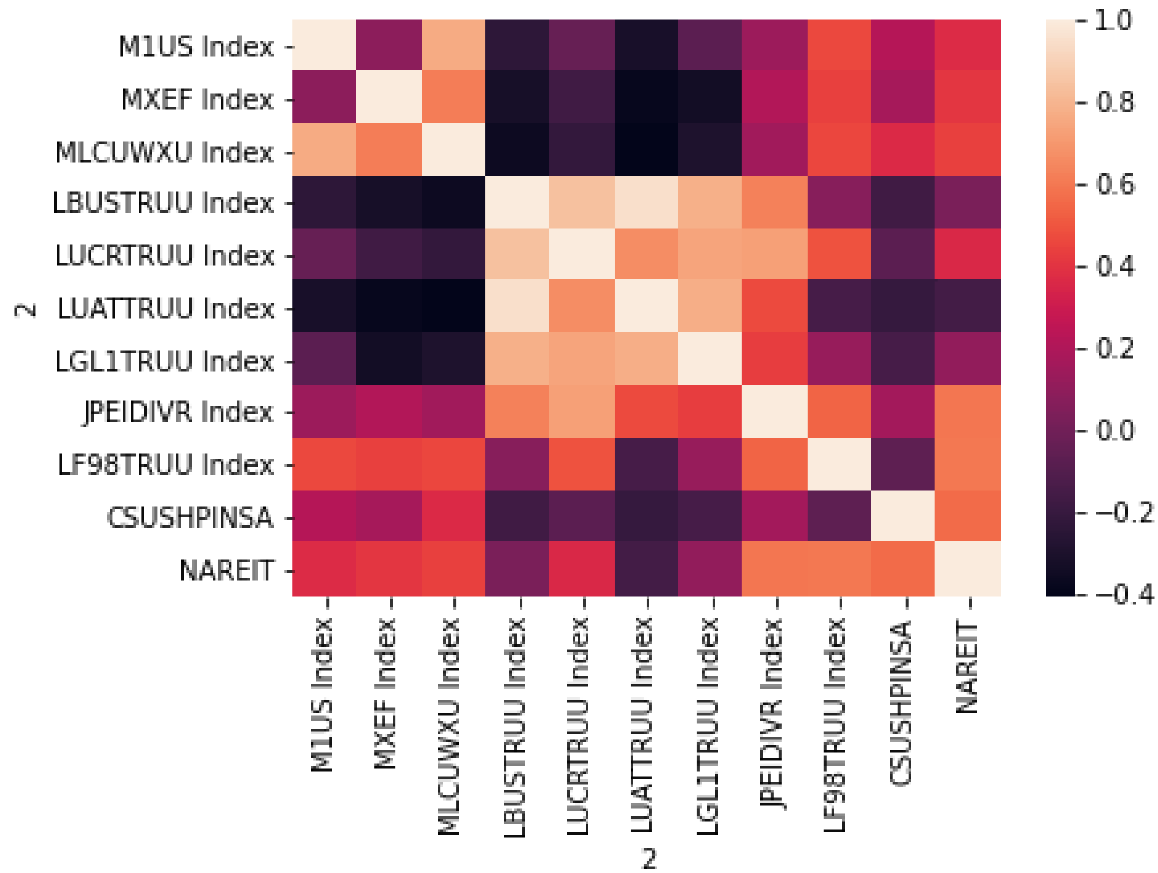
[37]:
```python
heatMap1 = sns.heatmap(ret3MLog.corr())
#plt.savefig("heatmap3M.png")
```




```
[38]: heatMap1 = sns.heatmap(ret1YLog.corr())
      plt.savefig("heatmap1Y.png")
```



[39]:
```
heatMap1 = sns.heatmap(ret3YLog.corr())
plt.savefig("heatmap3Y.png")
```



Let's use the 1Y return, volatility and correlation!

[40]:
```
def portfolio_annualised_performance(weights, mean_returns, cov_matrix):
    returns = np.sum(mean_returns*weights) *252
    std = np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights))) * np.sqrt(252)
    return std, returns

def random_portfolios(num_portfolios, mean_returns, cov_matrix, risk_free_rate):
    results = np.zeros((3,num_portfolios))
    weights_record = []
    for i in range(num_portfolios):
        weights = np.random.random(4)
        weights /= np.sum(weights)
        weights_record.append(weights)
        portfolio_std_dev, portfolio_return =␣
 ↪portfolio_annualised_performance(weights, mean_returns, cov_matrix)
        results[0,i] = portfolio_std_dev
        results[1,i] = portfolio_return
        results[2,i] = (portfolio_return - risk_free_rate) / portfolio_std_dev
    return results, weights_record
```

```
def display_simulated_ef_with_random(mean_returns, cov_matrix, num_portfolios,␣
↪risk_free_rate):
    results, weights = random_portfolios(num_portfolios,mean_returns,␣
↪cov_matrix, risk_free_rate)

    max_sharpe_idx = np.argmax(results[2])
    sdp, rp = results[0,max_sharpe_idx], results[1,max_sharpe_idx]
    max_sharpe_allocation = pd.DataFrame(weights[max_sharpe_idx],index=table.
↪columns,columns=['allocation'])
    max_sharpe_allocation.allocation = [round(i*100, 2)for i in␣
↪max_sharpe_allocation.allocation]
    max_sharpe_allocation = max_sharpe_allocation.T

    min_vol_idx = np.argmin(results[0])
    sdp_min, rp_min = results[0,min_vol_idx], results[1,min_vol_idx]
    min_vol_allocation = pd.DataFrame(weights[min_vol_idx],index=table.
↪columns,columns=['allocation'])
    min_vol_allocation.allocation = [round(i*100,2)for i in min_vol_allocation.
↪allocation]
    min_vol_allocation = min_vol_allocation.T

    print("-"*80)
    print("Maximum Sharpe Ratio Portfolio Allocation\n")
    print("Annualised Return:", round(rp, 2))
    print("Annualised Volatility:", round(sdp, 2))
    print("\n")
    print(max_sharpe_allocation)
    print("-"*80)
    print("Minimum Volatility Portfolio Allocation\n")
    print("Annualised Return:", round(rp_min, 2))
    print("Annualised Volatility:", round(sdp_min, 2))
    print("\n")
    print(min_vol_allocation)

    plt.figure(figsize=(10, 7))
    plt.scatter(results[0,:],results[1,:],c=results[2,:],cmap='YlGnBu',␣
↪marker='o', s=10, alpha=0.3)
    plt.colorbar()
    plt.scatter(sdp,rp,marker='*',color='r',s=500, label='Maximum Sharpe ratio')
    plt.scatter(sdp_min,rp_min,marker='*',color='g',s=500, label='Minimum␣
↪volatility')
    plt.title('Simulated Portfolio Optimization based on Efficient Frontier')
    plt.xlabel('annualised volatility')
    plt.ylabel('annualised returns')
    plt.legend(labelspacing=0.8)
```

```
#display_simulated_ef_with_random(list(AnnualisedRet['Ret1YAnnual']),
#                                  np.cov(ret1YLog.T),
#                                  22,
#                                 0.002)
```

Dropping Case Shiller from the dataset below

[41]:
```
stocks_to_be_dropped = ['LUCRTRUU Index','LGL1TRUU Index', 'MLCUWXU Index']
ret1YLog_NEW = ret1YLog.drop(stocks_to_be_dropped, axis=1)
ret1MAnnulaisedLog_NEW = ret1MAnnulaisedLog.drop(stocks_to_be_dropped, axis=1)
ret1MLog = ret1MLog.drop(stocks_to_be_dropped, axis=1)
annualRet_NEW = pd.DataFrame(AnnualisedRet['Ret1YAnnual']).
 ↪drop(stocks_to_be_dropped)
```

[42]:
```
ret1MLog_wo_cs = ret1MLog.drop('CSUSHPINSA', axis=1)
```

[43]:
```
#beta = 2.1
#alpha = 0.01
#ret1MAnnulaisedLog_NEW['Unison'] = beta * (ret1MLog['CSUSHPINSA'] + alpha)
```

[44]:
```
ind_er = (1 + ret1MLog_wo_cs.mean()) ** 12 - 1
cov_matrix = ret1MLog_wo_cs.cov()
```

6 MonteCarlo Implementation

[45]:
```
def montecarlo():
    p_ret = [] # Define an empty array for portfolio returns
    p_vol = [] # Define an empty array for portfolio volatility
    p_weights = [] # Define an empty array for asset weights

    num_assets = len(ret1MLog_wo_cs.columns)
    num_portfolios = 50000

    for portfolio in range(num_portfolios):
        weights = np.random.random(num_assets)
        weights = weights/np.sum(weights)
        p_weights.append(weights)
        returns = np.dot(weights, ind_er) # Returns are the product of␣
 ↪individual expected returns of asset and its
                                          # weights
        p_ret.append(returns)
        var = cov_matrix.mul(weights, axis=0).mul(weights, axis=1).sum().sum()#␣
 ↪Portfolio Variance
        sd = np.sqrt(var) # Daily standard deviation
        ann_sd = sd*np.sqrt(12) # Annual standard deviation = volatility
        p_vol.append(ann_sd)
```

```
    data = {'Returns':p_ret, 'Volatility':p_vol}

    for counter, symbol in enumerate(ret1MLog_wo_cs.columns.tolist()):
        #print(counter, symbol)
        data[symbol+' weight'] = [w[counter] for w in p_weights]

    portfolios  = pd.DataFrame(data)
    display(portfolios.head()) # Dataframe of the 10000 portfolios created

    #Plot efficient frontier
    #print("Efficient Frontier::")
    #portfolios.plot.scatter(x='Volatility', y='Returns', marker='o', s=10,␣
↪alpha=0.3, grid=True, figsize=[10,10])

    min_vol_port = portfolios.iloc[portfolios['Volatility'].idxmin()]
    # idxmin() gives us the minimum value in the column specified.             ␣
↪
    print("Min Vol Portfolio::")
    display(min_vol_port)

    rf = 0.02 # risk factor
    optimal_risky_port = portfolios.iloc[((portfolios['Returns']-rf)/
↪portfolios['Volatility']).idxmax()]
    display(optimal_risky_port)
    print("Max Sharpe Ratio:", max((portfolios['Returns']-rf)/
↪portfolios['Volatility']))

    # Plotting min vol portfolio and optimal portfolio
    print("Efficient Frontier and Min Variance Portfolio and Max Sharpe Ratio␣
↪Portfolio::")
    fig, ax = plt.subplots(figsize=(10, 10))
    ax.set_ylim(ymin=0, ymax=0.1)
    ax.set_xlim(xmin=0, xmax=0.2)
    plt.scatter(portfolios['Volatility'], portfolios['Returns'],marker='o',␣
↪s=10, alpha=0.3)
    plt.scatter(min_vol_port[1], min_vol_port[0], color='r', marker='*', s=500)
    plt.scatter(optimal_risky_port[1], optimal_risky_port[0], color='g',␣
↪marker='*', s=500)
```

[46]:
```
montecarlo()
```

```
   Returns  Volatility  M1US Index weight  MXEF Index weight  \
0    5.96%       9.74%              4.86%             30.28%
1    7.85%       8.56%             13.95%              3.14%
2    6.04%       6.23%              1.83%             13.04%
3    7.43%      11.18%              0.71%             19.70%
4    7.45%      11.60%             20.97%             20.21%
```

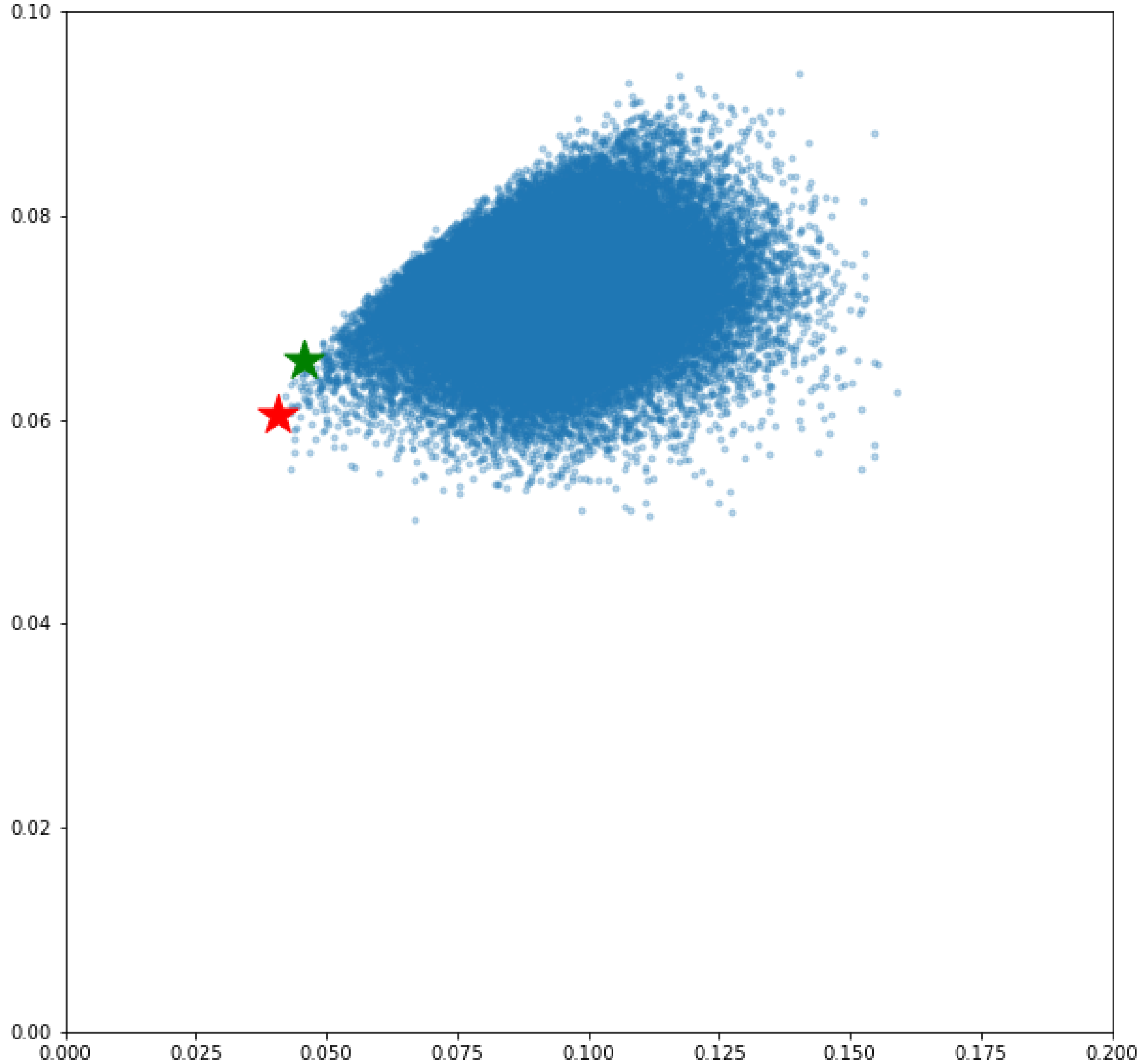
```
   LBUSTRUU Index weight  LUATTRUU Index weight  JPEIDIVR Index weight  \
0                 12.12%                 23.76%                 10.56%
1                 14.04%                 20.14%                 23.36%
2                 23.02%                 28.14%                  1.32%
3                  8.23%                  7.89%                 27.58%
4                 12.63%                  5.72%                 10.60%

   LF98TRUU Index weight  NAREIT weight
0                 13.03%          5.38%
1                  2.35%         23.03%
2                 25.13%          7.50%
3                 12.23%         23.67%
4                  9.39%         20.48%
```

```
Min Vol Portfolio::
```

```
Returns                  6.04%
Volatility               4.07%
M1US Index weight        9.78%
MXEF Index weight        2.91%
LBUSTRUU Index weight   28.98%
LUATTRUU Index weight   44.98%
JPEIDIVR Index weight    5.47%
LF98TRUU Index weight    6.33%
NAREIT weight            1.55%
Name: 8661, dtype: float64
```

```
Returns                  6.59%
Volatility               4.55%
M1US Index weight       10.99%
MXEF Index weight        0.30%
LBUSTRUU Index weight   40.44%
LUATTRUU Index weight   24.77%
JPEIDIVR Index weight   12.85%
LF98TRUU Index weight    8.28%
NAREIT weight            2.38%
Name: 8863, dtype: float64
```

```
Max Sharpe Ratio: 1.0085199699760694
Efficient Frontier and Min Variance Portfolio and Max Sharpe Ratio Portfolio::
```



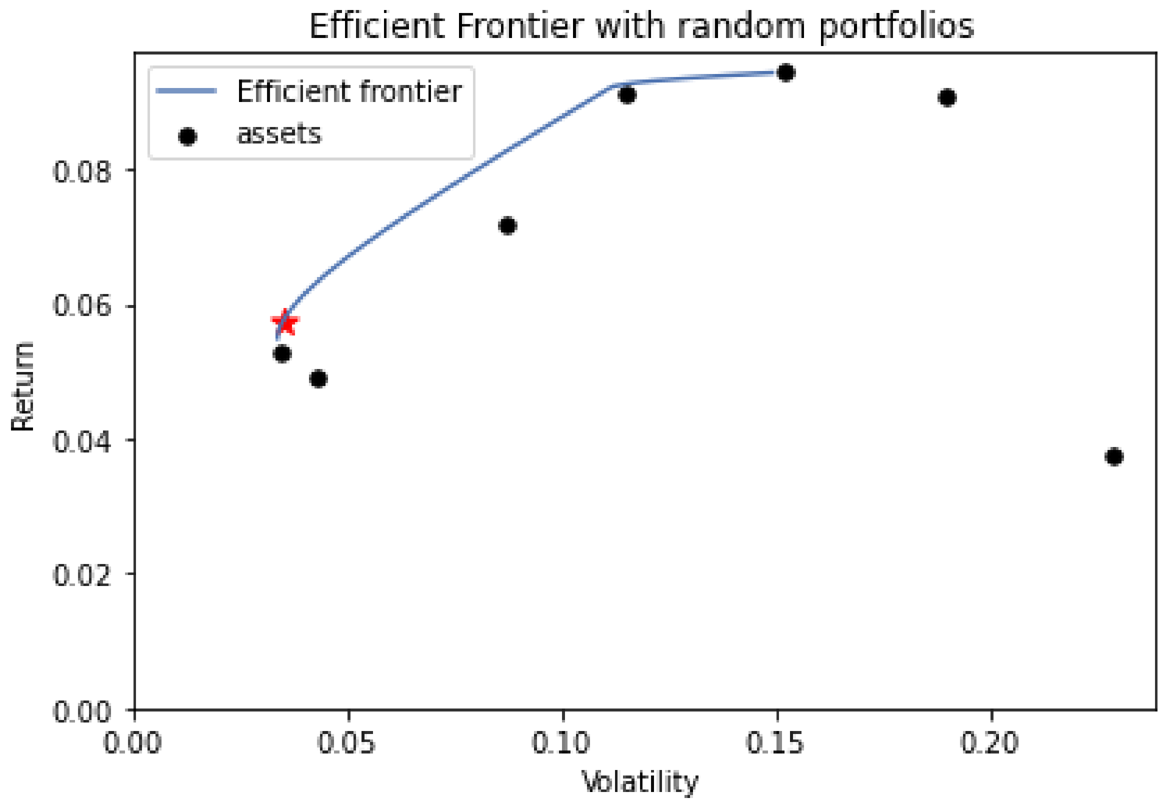
6.1 PyPfPortfolio Implementation

[47]:
```
def pyEffPortFlio(ret):
    from pypfopt import plotting
    import pandas as pd
    from pypfopt.efficient_frontier import EfficientFrontier
    from pypfopt import risk_models
    from pypfopt import expected_returns
    mu = expected_returns.mean_historical_return(ret, returns_data=True,␣
 ↪frequency=12, compounding=False)
    S = risk_models.sample_cov(ret, returns_data=True, frequency=12)

    # Optimize for maximal Sharpe ratio
    ef = EfficientFrontier(mu, S)
```

```
    #weights = ef.max_sharpe(risk_free_rate = factors['RF'].mean()*12/100)
    fig, ax = plt.subplots()
    #mkt_std = (mkt.std()*np.sqrt(12)/100)
    #mkt_ret = (mkt.mean()*12/100)
    #xvalues=[0,mkt_std]
    #yvalues=[0.04457,mkt_ret]
    #plt.plot(xvalues,yvalues)
    #xvalues=[0,0.218]
    #yvalues=[0.04457,0.199]
    #plt.plot(xvalues,yvalues)
    plotting.plot_efficient_frontier(ef, ax=ax, show_assets=True)
    ef.max_sharpe()
    ret_tangent, std_tangent, _ = ef.portfolio_performance(verbose=True)
    ax.scatter(std_tangent, ret_tangent, marker="*", s=100, c="r", label="Max␣
↪Sharpe")

    # Output
    ax.set_title("Efficient Frontier with random portfolios")
    ax.set_ylim(ymin=0)
    ax.set_xlim(xmin=0)
    plt.show()
```

[48]:
```
pyEffPortFlio(ret1MLog_wo_cs)
```

```
Expected annual return: 5.8%
Annual volatility: 3.5%
Sharpe Ratio: 1.07
```



7 CVXOPT Implementation

[49]:
```
import numpy as np
import cvxopt as opt
from cvxopt import blas, solvers
import matplotlib.pyplot as plt
```

[50]:
```
def optimal_portfolio(returns):
    # Turn off progress printing
    solvers.options['show_progress'] = False
    returns = np.asmatrix(returns.T)                # -> (n_assets,␣
↪n_observations)
    n_assets = len(returns)
    # Vector of desired returns
    N = n_assets*(int(1e+2))
    mus = [10 ** (5.0 * t / N - 1.0) for t in range(N)]


    # Obtain expected returns and covariance
    m1 = np.mean(returns, axis=1)                       # Mean returns
```

```
    c1 = np.cov(returns, bias=True)                         # Volatility (in terms␣
↪of standard deviation)
    # Convert to cvxopt matrices
    pbar = opt.matrix(m1)
    S = opt.matrix(c1)

    # Limits for each stock
    lower_bound = 0.0 #5%
    upper_bound = 1
    # Check error
    if n_assets*lower_bound > 1:
        print('Too many stocks for the lower bound limit.')
        lower_bound = round(1.00/n_assets, 3)
        print('New lower band: ', lower_bound)

    upper_bound_array = np.ones((n_assets, 1)) * upper_bound


    # Create constraint matrices
    G = opt.matrix(np.vstack((-np.eye(n_assets), np.eye(n_assets))))
    #h = opt.matrix(np.vstack((-lower_bound*np.ones((n_assets, 1)),␣
↪upper_bound*np.ones((n_assets, 1)), 1.2*Benchmark_weights*np.ones((11, 1)))))
    h = opt.matrix(np.vstack((-lower_bound*np.ones((n_assets, 1)),␣
↪upper_bound_array)))
    A = opt.matrix(1.0, (1, n_assets))
    b = opt.matrix(1.0)

    # Calculate efficient frontier weights using quadratic programming
    portfolios = [solvers.qp(mu * S, -pbar, G, h, A, b)['x'] for mu in mus]
    sol = solvers.qp(S, -pbar, G, h, A, b)

    ## CALCULATE RISKS AND RETURNS FOR FRONTIER
    weights = [np.asarray(x) for x in portfolios]
    returns = [blas.dot(pbar, x) for x in portfolios]
    risks = [np.sqrt(blas.dot(x, S * x)) for x in portfolios]

    return weights, np.asarray(returns), np.asarray(risks), sol, returns, risks
```

7.1 Optimal Portfolio without Case-Shiller

[51]:
```
ret = ret1MLog_wo_cs
weights, returns, risks, sol, plot_return_wo_cs, plot_risk_wo_cs =␣
↪optimal_portfolio(ret)
ann_returns_wo_cs = (1 + returns)**12 - 1
ann_risks_wo_cs = np.sqrt(12) * risks
sharpe1 = (ann_returns_wo_cs - 0.02)/ann_risks_wo_cs
ind_opt = np.argmax(sharpe1)             # Index of selected portfolio
```

```python
opt_portfolio = {}
opt_portfolio['return'] = returns[ind_opt] * 12
opt_portfolio['risk'] = risks[ind_opt] * np.sqrt(12)
opt_portfolio['sharpe'] = sharpe1[ind_opt]

wt = weights[ind_opt]/sum(weights[ind_opt])
ind_w = np.flip(np.argsort(wt, axis=0), axis=0)
opt_portfolio['weights'] = wt[ind_w]
ind_w = ind_w.ravel().tolist()
sym1 = pd.DataFrame(list(ret))

sym=sym1.loc[ind_w]

#sym = [str(sym[k][0][0]) for k in range(len(sym))]
opt_portfolio['stocks'] = sym

output = pd.DataFrame(columns=["Ticker","Weights%"])
output["Ticker"] = sym[0]
output["Weights%"] = wt[ind_w]
output = output.reset_index(drop=True)
display(output)
print(opt_portfolio)
```

```
          Ticker  Weights%
0  LBUSTRUU Index    85.19%
1      M1US Index     7.85%
2  LF98TRUU Index     5.97%
3  JPEIDIVR Index     0.99%
4  LUATTRUU Index     0.00%
5          NAREIT     0.00%
6      MXEF Index     0.00%
```

```
{'return': 0.05756605592770696, 'risk': 0.03517106208680689, 'sharpe':
1.1119782375301672, 'weights': array([[[8.51884331e-01]],

       [[7.85272883e-02]],

       [[5.96615896e-02]],

       [[9.91016888e-03]],

       [[1.57891218e-05]],

       [[6.14080622e-07]],

       [[2.18675002e-07]]]), 'stocks':                  0
2  LBUSTRUU Index
```

```
0      M1US Index
5  LF98TRUU Index
4  JPEIDIVR Index
3  LUATTRUU Index
6          NAREIT
1      MXEF Index}
```

7.2 Optimal Portfolio with Case-Shiller

[52]:
```
ret = ret1MLog #ret1MLog*12
weights, returns, risks, sol, plot_return_cs, plot_risk_cs =␣
↪optimal_portfolio(ret)
ann_returns_cs = (1 + returns)**12 - 1
ann_risks_cs = np.sqrt(12) * risks
sharpe1 = (ann_returns_cs - 0.02)/ann_risks_cs
ind_opt = np.argmax(sharpe1)              # Index of selected portfolio

opt_portfolio = {}
opt_portfolio['return'] = returns[ind_opt] * 12
opt_portfolio['risk'] = risks[ind_opt] * np.sqrt(12)
opt_portfolio['sharpe'] = sharpe1[ind_opt]

wt = weights[ind_opt]/sum(weights[ind_opt])
ind_w = np.flip(np.argsort(wt, axis=0), axis=0)
opt_portfolio['weights'] = wt[ind_w]
ind_w = ind_w.ravel().tolist()
sym1 = pd.DataFrame(list(ret))

sym=sym1.loc[ind_w]

#sym = [str(sym[k][0][0]) for k in range(len(sym))]
opt_portfolio['stocks'] = sym

output = pd.DataFrame(columns=["Ticker","Weights%"])
output["Ticker"] = sym[0]
output["Weights%"] = wt[ind_w]
output = output.reset_index(drop=True)
display(output)
print(opt_portfolio)
```

	Ticker	Weights%
0	CSUSHPINSA	54.28%
1	LBUSTRUU Index	41.12%
2	M1US Index	2.89%
3	LF98TRUU Index	0.93%
4	JPEIDIVR Index	0.78%
5	LUATTRUU Index	0.00%
6	NAREIT	0.00%

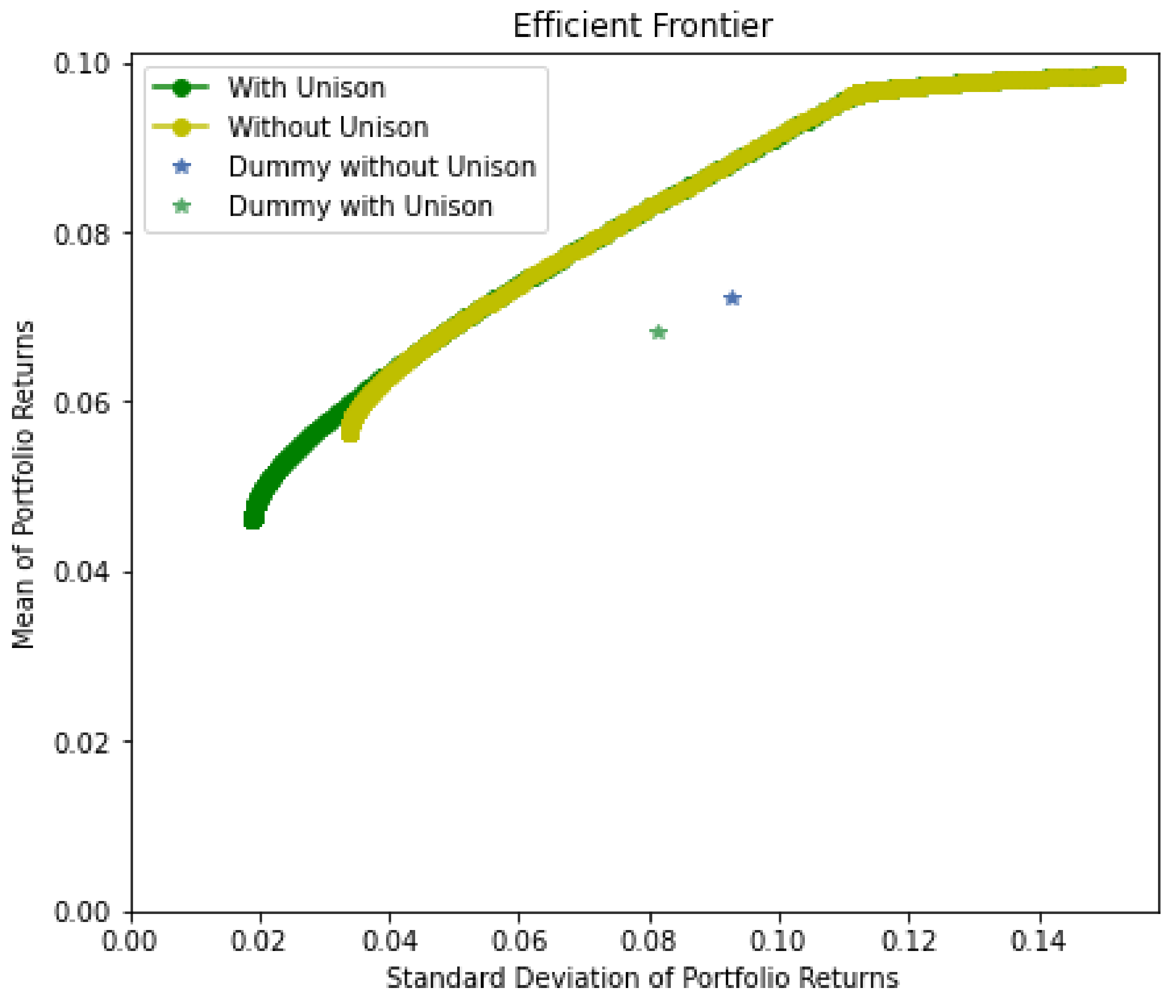
```
7      MXEF Index     0.00%
```

```
{'return': 0.04808291963245642, 'risk': 0.020157701669695136, 'sharpe':
1.4464373588108888, 'weights': array([[[5.42797510e-01]],

       [[4.11185230e-01]],

       [[2.89008908e-02]],

       [[9.30195696e-03]],

       [[7.80931010e-03]],

       [[3.38432855e-06]],

       [[1.12253013e-06]],

       [[5.94927122e-07]]]), 'stocks':                 0
6      CSUSHPINSA
2  LBUSTRUU Index
0      M1US Index
5  LF98TRUU Index
4  JPEIDIVR Index
3  LUATTRUU Index
7          NAREIT
1      MXEF Index}
```

7.3 Efficient Frontier plots for comparision

[53]:
```
#Plot Efficient Frontier
weights = np.ones((1,7)) * 1/7
returns = np.dot(weights, ind_er)
risk = np.sqrt(weights @ ret1MLog_wo_cs.cov() @ weights.T).values *np.sqrt(12)
ind_er_uni = (1 + ret1MLog.mean())**12 - 1
weights = np.ones((1,8)) * 1/8
returns_uni = np.dot(weights, ind_er_uni)
risk_uni = np.sqrt(weights @ ret1MLog.cov() @ weights.T).values *np.sqrt(12)
fig, ax = plt.subplots(figsize=(7,6))
plt.plot(ann_risks_cs, ann_returns_cs, 'g-o', label='With Unison')
plt.plot(ann_risks_wo_cs, ann_returns_wo_cs, 'y-o', label='Without Unison')
plt.plot(risk,returns,'*', label='Dummy without Unison')
plt.plot(risk_uni,returns_uni,'*', label='Dummy with Unison')
plt.title('Efficient Frontier')
plt.ylabel('Mean of Portfolio Returns')
plt.xlabel('Standard Deviation of Portfolio Returns')
ax.set_ylim(ymin=0)
ax.set_xlim(xmin=0)
plt.legend()
```

```
plt.show()
```



7.3.1 Sharpe of Dummy Portfolio without Unison

[54]:
```
returns/risk
```

[54]: array([[0.78009566]])

7.3.2 Sharpe of Dummy Portfolio with Unison

[55]:
```
returns_uni/risk_uni
```

[55]: array([[0.83990393]])

9 Reference

[1] Streamlit package: https://docs.streamlit.io/en/stable/
[2] CVXOPT package: https://cvxopt.org/
[3] pyportfolioopt package: https://pypi.org/project/pyportfolioopt/
[4] Long Term Capital Market Assumptions: https://am.jpmorgan.com/content/dam/jpm-am-aem/global/en/insights/portfolio-insights/ltcma/ltcma-full-report.pdf