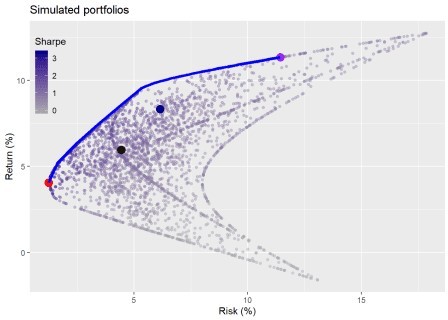
…Sharpe ratio portfolios. We found that you can shoot for high returns or high risk-adjusted returns, but rarely both. Assuming no major change in the underlying average returns and risk, choosing the efficient high return or high risk-adjusted return portfolio generally leads to similar performance a majority of the time in out-of-sample simulations. What was interesting about the results was that even though we weren’t arguing that one should favor satisficing over optimizing, we found that the satisfactory portfolio generally performed well.

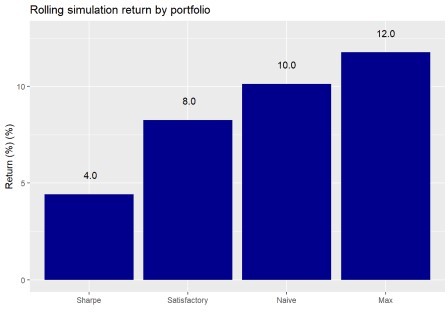
One area we didn’t test was the effect of sequence on mean-variance optimization. Recall we simulated 1,000 different sixty month (five year) return profiles for four potential assets—stocks, bonds, commodities (gold) and real estate. The weights we used to calculate the different portfolio returns and risk were based on data from 1987-1991 and were kept constant for all the simulations. What if the weighting system could “learn” from previous scenarios or adjust to recent information?

In this post, we’ll look at the impact of incorporating sequential information on portfolio results and compare that to the non-sequential results. To orient folks, we’ll show the initial portfolio weight simulation graph with the different portfolios and efficient frontier based on the historical data. The satisfactory, naive, MVO-derived maximum Sharpe ratio, and MVO-derived maximum return portfolios are the colored blue, black, red, and purple points, respectively.

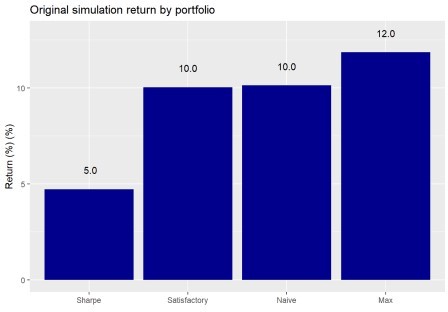


Now we’ll take the weighting schemes and apply them to the return simulations. However, we’ll run through the returns sequentially and calculate the returns and risk-adjusted returns for the weighting scheme using the prior simulation as the basis for the allocation on the next simulation. For example, we’ll calculate the efficient frontier and run the portfolio weight algorithm on simulation #75. From those calculations, we’ll assign the weights for the satisfactory, maximum Sharpe ratio, and maximum efficient return portfolios to compute the returns and risk-adjusted returns on simulation #76. For the record, simulation #1 uses the same weights as those that produced the dots in the graph above.

Running the simulations, here’s the average return by weighting algorithm.



And here’s the original.

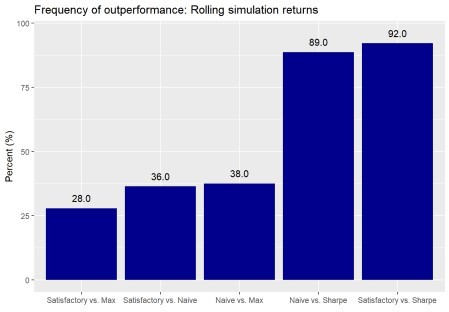


For the naive and maximum return portfolios, the average return isn’t much different on the sequential calculations than on original constant weight computations. The sequential Sharpe allocations are about 1% point lower than the stable ones. However, the sequential satisfactory allocations are almost 2% points lower than the constant allocation. In about 5% of the cases, the return simulation was not capable of achieving a satisfactory portfolio with the original constraints of not less than 7% and not more than 10% annualized return and risk. In such a case, we reverted to the original weighting. This adjustment did not affect the average return even when we excluded the “unsatisfactory” portfolios.

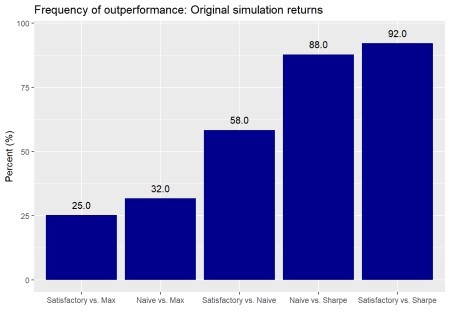
We assume that the performance drag is due to the specificity of the constraints amid random outcomes. The other portfolios (except for the naive one) optimize for the previous scenario and apply those optimal weights on the next portfolio in the sequence. The satisfactory portfolio chooses average weights to achieve a particular constraint and then applies those weights in sequence. Due to the randomness of the return scenarios, it seems possible that a weighting scheme that works for one scenario would produce poor results on another scenario. Of course, the satisfactory portfolios still achieved their return constraint on average.

This doesn’t exactly explain why the MVO portfolios didn’t suffer a performance drag. Our intuition is that the MVO portfolios are border cases. Hence, there probably won’t be too much dispersion in the results on average since the return simulations draw from the same return, risk, and error terms throughout. However, the satisfactory portfolio lies with the main portion of the territory, so the range of outcomes could be much larger. Clearly, this requires further investigation, but we’ll have to shelve that for now.

In the next set of graphs, we check out how each portfolio performs relative to the others. First the sequential simulation.



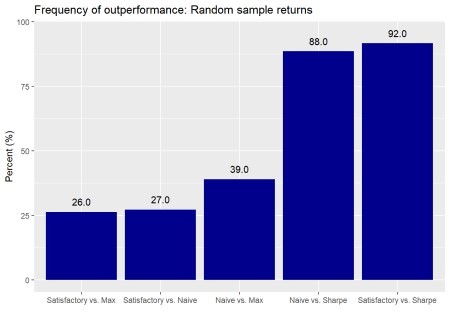
And the original:



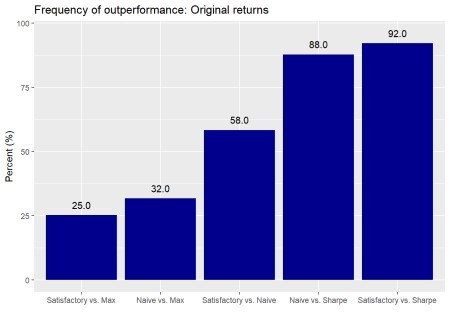
There are some modest changes in the satisfactory and naive vs. MVO-derived portfolios. But the biggest change occurs in the satisfactory vs. naive portfolio—almost a 22 percentage point decline! This appears to support our conjecture above that the order of the sequence could be affecting the satisfactory portfolio’s performance. Nonetheless, sequencing does not appear to alter the performance of the naive and satisfactory portfolios relative to the MVO-derived ones in a material way. Let’s randomize the sequence to say if results change.

For the random draw case, we randomly draw one return simulation on which to calculate our portfolio weights and then apply those weights on another randomly drawn portfolio. The draws are without replacement to ensure we don’t use the same simulation both to calculate weights and returns as well as to ensure that we use all simulations.

The average return graph isn’t much different for the random sampling than it is for the sequential so we’ll save some space and not print it. Here is the relative performance graph.



And here is the original relative performance for ease of comparison.



With random ordering we see that the naive and satisfactory portfolios relative to the maximum return experience a modest performance improvement. Relative to the maximum Sharpe portfolio, the performance is relatively unchanged. The satisfactory portfolio again suffers a drag relative to the naive, but somewhat less severe that in the sequential case.

What are the takeaways? In both sequential and random sampling, the performance of the naive and satisfactory compared to the MVO portfolios was relatively stable. That supports our prior observation that you can shoot for high returns or high risk-adjusted returns, but usually not

both. It’s also noteworthy that the maximum Sharpe ratio consistently underperforms the naive and satisfactory portfolios in all of the simulations. Here, part of the study found that the maximum Sharpe portfolio suffered the worst performance relative to many others including an equal- weighted (i.e., naive) portfolio.

One issue with our random sampling test is that it was only one random sequence. To be truly robust, we’d want to run the sampling simulation more than once. But we’ll have to save that for when we rent time on a cloud GPU because we don’t think our flintstone-aged laptop will take kindly to running 1,000 simulations of 1,000 random sequences of 1,000 simulated portfolios with 1,000 optimizations and 3,000 random portfolio weightings!

Perceptive readers might quibble with how we constructed our “learning” simulations. Indeed, in both the sequential and random sample cases we calculated our allocation weights only using the prior simulation in the queue (e.g. weights derived from return simulation #595 are deployed on simulation #596 (or, for example, #793 in the random simulation)). But that “learning” could be considered relatively short term. A more realistic scenario would be to allow for cumulative learning by trying to incorporate all or a majority of past returns in the sequence. And we could get even more complicated by throwing in some weighting scheme to emphasize nearer term results. Finally, we could calculate cumulative returns over one scenario or multiple scenarios. We suspect relative performance would be similar, but you never know!

Until we experiment with these other ideas, the R and Python code are below. While we generally don’t discuss the code in detail, we have been getting more questions and constructive feedback on it of late. Thank you for that! One thing to note is the R code to produce the simulations runs much quicker than the Python code. Part of that is likely due to how we coded in the efficient frontier and maximum Sharpe and return portfolios in the different languages. But the differences are close to an order of magnitude. If anyone notices anything we could do to improve performance, please drop us an email at nbw dot osm at gmail dot com. Thanks!

# R code

# Built using R 3.6.2 ### Load packages

suppressPackageStartupMessages({ library(tidyquant) library(tidyverse)

})

### Load data

# Seem prior posts for how we built these data frames df <- readRDS("port\_const.rds")

dat <- readRDS("port\_const\_long.rds")

sym\_names <- c("stock", "bond", "gold", "realt", "rfr") sim1 <- readRDS("hist\_sim\_port16.rds")

### Call functions

# See prior posts for how we built these functions source("Portfolio\_simulation\_functions.R") source("Efficient\_frontier.R")

## Function for calculating satisfactory weighting

port\_sim\_wts <- function(df, sims, cols, return\_min, risk\_max){

if(ncol(df) != cols){ print("Columns don't match") break

}

# Create weight matrix

wts <- matrix(nrow = (cols-1)\*sims, ncol = cols) count <- 1

for(i in 1:(cols-1)){ for(j in 1:sims){

a <- runif((cols-i+1),0,1) b <- a/sum(a)

c <- sample(c(b,rep(0,i-1))) wts[count,] <- c

count <- count+1

}

}

# Find returns

mean\_ret <- colMeans(df)

# Calculate covariance matrix cov\_mat <- cov(df)

# Calculate random portfolios

port <- matrix(nrow = (cols-1)\*sims, ncol = 2) for(i in 1:nrow(port)){

port[i,1] <- as.numeric(sum(wts[i,] \* mean\_ret))

port[i,2] <- as.numeric(sqrt(t(wts[i,]) %\*% cov\_mat %\*% wts[i,]))

}

port <- as.data.frame(port) %>%

`colnames<-`(c("returns", "risk"))

port\_select <- cbind(port, wts) port\_wts <- port\_select %>%

mutate(returns = returns\*12, risk = risk\*sqrt(12)) %>%

filter(returns >= return\_min, risk <= risk\_max) %>%

summarise\_at(vars(3:6), mean) port\_wts

}

## Portfolio func

port\_func <- function(df,wts){ mean\_ret = colMeans(df) returns = sum(mean\_ret\*wts)

risk = sqrt(t(wts) %\*% cov(df) %\*% wts) c(returns, risk)

}

## Portfolio graph

pf\_graf <- function(df, nudge, multiplier, rnd, y\_lab, text){ df %>%

gather(key, value) %>%

ggplot(aes(reorder(key, value), value\*multiplier\*100)) + geom\_bar(stat='identity',

fill = 'darkblue') +

geom\_text(aes(label = format(round(value\*multiplier,rnd)\*100,nsmall

= 1)), nudge\_y = nudge)+ labs(x = "",

y = paste(y\_lab, "(%)", sep = " "),

title = paste(text, "by portfolio", sep = " "))

}

## Create outperformance graph

perf\_graf <- function(df, rnd, nudge, text){ df %>%

rename("Satisfactory vs. Naive" = ovs, "Satisfactory vs. Max" = ovr, "Naive vs. Max" = rve, "Satisfactory vs. Sharpe" = ove, "Naive vs. Sharpe" = sve) %>%

gather(key, value) %>% ggplot(aes(reorder(key, value), value\*100)) + geom\_bar(stat='identity',

fill = 'darkblue') +

geom\_text(aes(label = format(round(value,rnd)\*100, nsmall = 1)), nudge\_y = nudge)+

labs(x = "",

y = "Percent (%)",

title = paste("Frequency of outperformance:", text, sep = "

"))

}

### Create weight schemes

satis\_wts <- c(0.32, 0.4, 0.08, 0.2) # Calculated in previous post using port\_select\_func

simple\_wts <- rep(0.25, 4)

eff\_port <- eff\_frontier\_long(df[2:61,2:5], risk\_increment = 0.01) eff\_sharp\_wts <- eff\_port[which.max(eff\_port$sharpe),1:4] %>%

as.numeric()

eff\_max\_wts <- eff\_port[which.max(eff\_port$exp\_ret), 1:4] %>% as.numeric()

## Run port sim on 1987-1991 data

port\_sim\_1 <- port\_sim\_lv(df[2:61,2:5],1000,4)

# Run function on three weighting schemes and one simulation weight\_list <- list(satis = satis\_wts,

naive = simple\_wts, sharp = eff\_sharp\_wts, max = eff\_max\_wts)

wts\_df <- data.frame(wts = c("satis", "naive", "sharp", "max"), returns

= 1:4, risk = 5:8,

stringsAsFactors = FALSE)

for(i in 1:4){

wts\_df[i, 2:3] <- port\_func(df[2:61,2:5], weight\_list[[i]])

}

wts\_df$sharpe = wts\_df$returns/wts\_df$risk

# Graph portfolio simulation with three portfolios port\_sim\_1$graph +

geom\_point(data = wts\_df,

aes(x = risk\*sqrt(12)\*100, y = returns\*1200), color = c("darkblue", "black", "red", "purple"), size = 4) +

geom\_line(data = eff\_port,

aes(stdev\*sqrt(12)\*100, exp\_ret\*1200), color = 'blue',

size = 1.5) +

theme(legend.position = c(0.05,0.8), legend.key.size = unit(.5, "cm"),

legend.background = element\_rect(fill = NA))

## Calculate performance with rolling frontier on max sharpe set.seed(123)

eff\_sharp\_roll <- list() eff\_max\_roll <- list() satis\_roll <- list() for(i in 1:1000){

if(i == 1){

sharp\_wts <- eff\_sharp\_wts max\_wts <- eff\_max\_wts sat\_wts <- satis\_wts

}else {

eff\_calc <- eff\_frontier\_long(sim1[[i-1]]$df, risk\_increment = 0.01)

sharp\_wts <- eff\_calc[which.max(eff\_calc$sharpe), 1:4] max\_wts <- eff\_calc[which.max(eff\_calc$exp\_ret), 1:4] sat\_wts <- port\_sim\_wts(sim1[[i-1]]$df, 1000, 4, 0.07, 0.1)

}

eff\_sharp\_roll[[i]] <- port\_func(sim1[[i]]$df, as.numeric(sharp\_wts)) eff\_max\_roll[[i]] <- port\_func(sim1[[i]]$df, as.numeric(max\_wts)) satis\_roll[[i]] <- port\_func(sim1[[i]]$df, as.numeric(sat\_wts))

}

list\_to\_df <- function(list\_ob){

x <- do.call('rbind', list\_ob) %>% as.data.frame() %>%

`colnames<-`(c("returns", "risk")) %>% mutate(sharpe = returns/risk)

x

}

roll\_lists <- c("eff\_sharp\_roll", "eff\_max\_roll", "satis\_roll")

for(obj in roll\_lists){

x <- list\_to\_df(get(obj)) assign(obj, x)

}

# Return

roll\_mean\_pf <- data.frame(Satisfactory = mean(satis\_roll[,1], na.rm = TRUE),

Naive = mean(simple\_df[,1]), Sharpe = mean(eff\_sharp\_roll[,1]), Max = mean(eff\_max\_roll[,1]))

# Graph mean returns

pf\_graf(roll\_mean\_pf, 1, 12, 2,"Return (%)", "Rolling simulation return")

pf\_graf(mean\_pf, 1, 12, 2,"Return (%)", "Original simulation return")

# Create relative performance df

roll\_ret\_pf <- data.frame(ovs = mean(satis\_df[,1] > simple\_df[,1]),

ovr = mean(satis\_df[,1] > eff\_max\_roll[,1]), rve = mean(simple\_df[,1] > eff\_max\_roll[,1]), ove = mean(satis\_df[,1] > eff\_sharp\_roll[,1]), sve = mean(simple\_df[,1] > eff\_sharp\_roll[,1]))

# Graph outperformance

perf\_graf(roll\_ret\_pf, 2, 4, "Rolling simulation returns")

perf\_graf(ret\_pf, 2, 4, "Original simulation returns")

## Sampling portfolios set.seed(123) eff\_sharp\_samp <- list() eff\_max\_samp <- list() satis\_samp <- list()

for(i in 1:1000){

if(i == 1){

sharp\_wts <- eff\_sharp\_wts max\_wts <- eff\_max\_wts sat\_wts <- satis\_wts

}else {

samp\_1 <- sample(1000, 1) # Sample a return simulation for weight scheme

eff\_calc <- eff\_frontier\_long(sim1[[samp\_1]]$df, risk\_increment = 0.01)

sharp\_wts <- eff\_calc[which.max(eff\_calc$sharpe), 1:4] max\_wts <- eff\_calc[which.max(eff\_calc$exp\_ret), 1:4]

sat\_wts <- port\_sim\_wts(sim1[[samp\_1]]$df, 1000, 4, 0.07, 0.1)

}

samp\_2 <- sample(1000, 1) # sample a return simulation to analyze performance

eff\_sharp\_samp[[i]] <- port\_func(sim1[[samp\_2]]$df, as.numeric(sharp\_wts))

eff\_max\_samp[[i]] <- port\_func(sim1[[samp\_2]]$df, as.numeric(max\_wts))

satis\_samp[[i]] <- port\_func(sim1[[samp\_2]]$df, as.numeric(sat\_wts))

}

samp\_lists <- c("eff\_sharp\_samp", "eff\_max\_samp", "satis\_samp") for(obj in samp\_lists){

x <- list\_to\_df(get(obj)) assign(obj, x)

}

# Return

samp\_mean\_pf <- data.frame(Satisfactory = mean(satis\_samp[,1], na.rm=TRUE),

Naive = mean(simple\_df[,1]), Sharpe = mean(eff\_sharp\_samp[,1]), Max = mean(eff\_max\_samp[,1]))

# Graph mean returns: NOT SHOWN

pf\_graf(samp\_mean\_pf, 1, 12, 2,"Return (%)", "Random sampling return")

# pf\_graf(mean\_pf, 1, 12, 2,"Return (%)", "Original return")

# Create relative performance df

samp\_ret\_pf <- data.frame(ovs = mean(satis\_df[,1] > simple\_df[,1]),

ovr = mean(satis\_df[,1] > eff\_max\_samp[,1]), rve = mean(simple\_df[,1] > eff\_max\_samp[,1]), ove = mean(satis\_df[,1] >

eff\_sharp\_samp[,1]), eff\_sharp\_samp[,1]))

sve = mean(simple\_df[,1] >

# Graph outperformance

perf\_graf(samp\_ret\_pf, 2, 4, "Random sample returns")

perf\_graf(ret\_pf, 2, 4, "Original returns")

# Python code

# Built using Python 3.7.4 # Load libraries

import pandas as pd

import pandas\_datareader.data as web import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline plt.style.use('ggplot')

## Load data

# Seem prior posts for how we built these data frames df = pd.read\_pickle('port\_const.pkl')

dat = pd.read\_pickle('data\_port\_const.pkl') port\_names = ['Original','Naive', 'Sharpe', 'Max'] sim1 = pd.read\_pickle('hist\_sim\_port16.pkl')

## Load functions part 1

# Portfolio simulation functions

# NOte the Port\_sim class is slightly different than previous posts, # so we're reproducing it here.

# We were getting a lot of "numpy.float64 is not callable" errors

# due to overlapping names on variables and functions, so we needed # to fix the code. If it still throws error, let us know if it does.

## Simulation function class Port\_sim:

def calc\_sim(df, sims, cols): wts = np.zeros((sims, cols))

for i in range(sims):

a = np.random.uniform(0,1,cols)

b = a/np.sum(a) wts[i,] = b

mean\_ret = df.mean() port\_cov = df.cov()

port = np.zeros((sims, 2)) for i in range(sims):

port[i,0] = np.sum(wts[i,]\*mean\_ret)

port[i,1] = np.sqrt(np.dot(np.dot(wts[i,].T,port\_cov),

wts[i,]))

sharpe = port[:,0]/port[:,1]\*np.sqrt(12) return port, wts, sharpe

def calc\_sim\_lv(df, sims, cols):

wts = np.zeros(((cols-1)\*sims, cols)) count=0

for i in range(1,cols): for j in range(sims):

a = np.random.uniform(0,1,(cols-i+1)) b = a/np.sum(a)

c = np.random.choice(np.concatenate((b, np.zeros(i))),cols, replace=False)

wts[count,] = c count+=1

mean\_ret = df.mean() port\_cov = df.cov()

port = np.zeros(((cols-1)\*sims, 2)) for i in range(sims):

port[i,0] = np.sum(wts[i,]\*mean\_ret)

port[i,1] = np.sqrt(np.dot(np.dot(wts[i,].T,port\_cov),

wts[i,]))

sharpe = port[:,0]/port[:,1]\*np.sqrt(12) return port, wts, sharpe

def graph\_sim(port, sharpe): plt.figure(figsize=(14,6))

plt.scatter(port[:,1]\*np.sqrt(12)\*100, port[:,0]\*1200, marker='.', c=sharpe, cmap='Blues'

plt.colorbar(label='Sharpe ratio', orientation = 'vertical', shrink = 0.25)

plt.title('Simulated portfolios', fontsize=20) plt.xlabel('Risk (%)')

plt.ylabel('Return (%)') plt.show()

# Constraint function

def port\_select\_func(port, wts, return\_min, risk\_max):

port\_select = pd.DataFrame(np.concatenate((port, wts), axis=1)) port\_select.columns = ['returns', 'risk', 1, 2, 3, 4]

port\_wts = port\_select[(port\_select['returns']\*12 >= return\_min) & (port\_select['risk']\*np.sqrt(12) <= risk\_max)]

port\_wts = port\_wts.iloc[:,2:6] port\_wts = port\_wts.mean(axis=0) return port\_wts

def port\_select\_graph(port\_wts): plt.figure(figsize=(12,6))

key\_names = {1:"Stocks", 2:"Bonds", 3:"Gold", 4:"Real estate"} lab\_names = []

graf\_wts = port\_wts.sort\_values()\*100

for i in range(len(graf\_wts)):

name = key\_names[graf\_wts.index[i]] lab\_names.append(name)

plt.bar(lab\_names, graf\_wts, color='blue') plt.ylabel("Weight (%)")

plt.title("Average weights for risk-return constraint", fontsize=15)

for i in range(len(graf\_wts)): plt.annotate(str(round(graf\_wts.values[i])), xy=(lab\_names[i],

graf\_wts.values[i]+0.5))

plt.show()

## Load functions part 2

# We should have wrapped the three different efficient frontier functions

# into one class or function but ran out of time. This is probably what slows

# down the simulations below.

# Create efficient frontier function from scipy.optimize import minimize

def eff\_frontier(df\_returns, min\_ret, max\_ret): n = len(df\_returns.columns)

def get\_data(weights):

weights = np.array(weights)

returns = np.sum(df\_returns.mean() \* weights)

risk = np.sqrt(np.dot(weights.T, np.dot(df\_returns.cov(),

weights)))

sharpe = returns/risk

return np.array([returns,risk,sharpe])

# Contraints

def check\_sum(weights):

return np.sum(weights) - 1

# Rante of returns

mus = np.linspace(min\_ret,max\_ret,21)

# Function to minimize

def minimize\_volatility(weights): return get\_data(weights)[1]

# Inputs

init\_guess = np.repeat(1/n,n) bounds = ((0.0,1.0),) \* n

eff\_risk = [] port\_weights = []

for mu in mus:

# function for return

cons = ({'type':'eq','fun': check\_sum},

{'type':'eq','fun': lambda w: get\_data(w)[0] - mu})

result = minimize(minimize\_volatility, init\_guess,method='SLSQP',bounds=bounds,constraints=cons)

eff\_risk.append(result['fun']) port\_weights.append(result.x)

eff\_risk = np.array(eff\_risk) return mus, eff\_risk, port\_weights

# Create max sharpe function

from scipy.optimize import minimize def max\_sharpe(df\_returns):

n = len(df\_returns.columns)

def get\_data(weights):

weights = np.array(weights)

returns = np.sum(df\_returns.mean() \* weights)

risk = np.sqrt(np.dot(weights.T, np.dot(df\_returns.cov(), weights)))

sharpe = returns/risk

return np.array([returns,risk,sharpe])

# Function to minimize def neg\_sharpe(weights):

return -get\_data(weights)[2]

# Inputs

init\_guess = np.repeat(1/n,n) bounds = ((0.0,1.0),) \* n

# function for return

constraint = {'type':'eq','fun': lambda x: np.sum(x) - 1}

result = minimize(neg\_sharpe,

init\_guess, method='SLSQP', bounds=bounds, constraints=constraint)

return -result['fun'], result['x']

# Create efficient frontier function from scipy.optimize import minimize

def max\_ret(df\_returns):

n = len(df\_returns.columns) def get\_data(weights):

weights = np.array(weights)

returns = np.sum(df\_returns.mean() \* weights)

risk = np.sqrt(np.dot(weights.T, np.dot(df\_returns.cov(), weights)))

sharpe = returns/risk

return np.array([returns,risk,sharpe])

# Function to minimize def port\_ret(weights):

return -get\_data(weights)[0]

# Inputs

init\_guess = np.repeat(1/n,n) bounds = ((0.0,1.0),) \* n

# function for return

constraint = {'type':'eq','fun': lambda x: np.sum(x) - 1}

result = minimize(port\_ret,

init\_guess, method='SLSQP', bounds=bounds, constraints=constraint)

return -result['fun'], result['x']

## Load functions part 3 ## Portfolio return

def port\_func(df, wts): mean\_ret = df.mean()

returns = np.sum(mean\_ret \* wts)

risk = np.sqrt(np.dot(wts, np.dot(df.cov(), wts))) return returns, risk

## Return and relative performance graph

def pf\_graf(names, values, rnd, nudge, ylabs, graf\_title):

df = pd.DataFrame(zip(names, values), columns = ['key', 'value']) sorted = df.sort\_values(by = 'value')

plt.figure(figsize = (12,6))

plt.bar('key', 'value', data = sorted, color='darkblue')

for i in range(len(names)): plt.annotate(str(round(sorted['value'][i], rnd)), xy =

(sorted['key'][i], sorted['value'][i]+nudge))

plt.ylabel(ylabs)

plt.title('{} performance by portfolio'.format(graf\_title)) plt.show()

## Create portfolio np.random.seed(123)

port\_sim\_1, wts\_1, sharpe\_1 = Port\_sim.calc\_sim(df.iloc[1: 61,0:4],1000,4)

# Create returns and min/max ranges df\_returns = df.iloc[1:61, 0:4] min\_ret = min(port\_sim\_1[:,0] max\_ret = max(port\_sim\_1[:,0]

# Find efficient portfolio

eff\_ret, eff\_risk, eff\_weights = eff\_frontier(df\_returns, min\_ret, max\_ret

eff\_sharpe = eff\_ret/eff\_risk

## Create weight schemes

satis\_wts = np.array([0.32, 0.4, 0.08, 0.2]) # Calculated in previous post using port\_select\_func

simple\_wts = np.repeat(0.25, 4)

eff\_sharp\_wts = eff\_weights[np.argmax(eff\_sharpe)] eff\_max\_wts = eff\_weights[np.argmax(eff\_ret)]

wt\_list = [satis\_wts, simple\_wts, eff\_sharp\_wts, eff\_max\_wts] wts\_df = np.zeros([4,3])

for i in range(4):

wts\_df[i,:2] = port\_func(df.iloc[1:61,0:4], wt\_list[i]) wts\_df[:,2] = wts\_df[:,0]/wts\_df[:,1]

## Graph portfolios plt.figure(figsize=(12,6))

plt.scatter(port\_sim\_1[:,1]\*np.sqrt(12)\*100, port\_sim\_1[:,0]\*1200, marker='.', c=sharpe\_1, cmap='Blues' plt.plot(eff\_risk\*np.sqrt(12)\*100,eff\_ret\*1200,'b--',linewidth=2)

col\_code = ['blue', 'black', 'red', 'purple'] for i in range(4):

plt.scatter(wts\_df[i,1]\*np.sqrt(12)\*100, wts\_df[i,0]\*1200, c = col\_code[i], s = 50)

plt.colorbar(label='Sharpe ratio', orientation = 'vertical', shrink = 0.25)

plt.title('Simulated portfolios', fontsize=20) plt.xlabel('Risk (%)')

plt.ylabel('Return (%)') plt.show()

## Create simplifed satisfactory portfolio finder function def port\_sim\_wts(df1, sims1, cols1, ret1, risk1):

pf, wt, \_ = Port\_sim.calc\_sim(df1, sims1, cols1) port\_wts = port\_select\_func(pf, wt, ret1, risk1) return port\_wts

## Run sequential simulation np.random.seed(123)

eff\_sharp\_roll = np.zeros([1000,3]) eff\_max\_roll = np.zeros([1000,3]) satis\_roll = np.zeros([1000,3])

for i in range(1000): if i == 0:

sharp\_weights = eff\_sharp\_wts max\_weights = eff\_max\_wts sat\_weights = satis\_wts

else:

\_, sharp\_wts = max\_sharpe(sim1[i-1][0])

\_, max\_wts = max\_ret(sim1[i-1][0]) # Running the optimizatin twice is probably slowing down the simulation

sharp\_weights = sharp\_wts max\_weights = max\_wts

test = port\_sim\_wts(sim[i-1][0], 1000, 4, 0.07,0.1)

if np.isnan(test): sat\_weights = satis\_wts

else:

sat\_weights = test

eff\_sharp\_roll[i,:2] = port\_func(sim1[i][0], sharp\_weights) eff\_max\_roll[i,:2] = port\_func(sim1[i][0], max\_weights) satis\_roll[i,:2] = port\_func(sim1[i][0], sat\_weights)

eff\_sharp\_roll[:,2] = eff\_sharp\_roll[:,0]/eff\_sharp\_roll[:,1] eff\_max\_roll[:,2] = eff\_max\_roll[:,0]/eff\_max\_roll[:,1] satis\_roll[:,2] = satis\_roll[:,0]/satis\_roll[:,1]

# Calculate simple returns simple\_df = np.zeros([1000,3])

for i in range(1000):

simple\_df[i,:2] = port\_func(sim1[i][0], simple\_wts) simple\_df[:,2] = simple\_df[:,0]/simple\_df[:,1]

## Add simulations to list and graph

roll\_sim = [satis\_roll, simple\_df, eff\_sharp\_roll, eff\_max\_roll] port\_means = []

for df in roll\_sim: port\_means.append(np.mean(df[:,0])\*1200)

port\_names = ['Satisfactory', 'Naive', 'Sharpe', 'Max']

# Sequential simulation

pf\_graf(port\_names, port\_means, 1, 0.5, 'Returns (%)', 'Rolling simulation return')

# Original simulation port\_means1 = []

for df in list\_df:

## Comparison charts

# Build names for comparison chart comp\_names= []

for i in range(4):

for j in range(i+1,4):

comp\_names.append('{} vs. {}'.format(port\_names[i], port\_names[j]))

# Calculate comparison values comp\_values = []

for i in range(4):

for j in range(i+1, 4):

comps =np.mean(roll\_sim[i][:,0] > roll\_sim[j][:,0]) comp\_values.append(comps)

# Sequential comparisons

pf\_graf(comp\_names[:-1], comp\_values[:-1], 2, 0.025, 'Frequency (%)', 'Rolling simulation frequency of')

port\_means1.append(np.mean(df[:,0])\*1200)

pf\_graf(port\_names, port\_means1, 1, 0.5, 'Returns (%)', 'Original simulation return')

# original comparisons

# Calculate comparison values comp\_values1 = []

for i in range(4):

for j in range(i+1, 4):

comps1 = np.mean(list\_df[i][:,0] > list\_df[j][:,0]) comp\_values1.append(comps1)

pf\_graf(comp\_names[:-1], comp\_values1[:-1], 2, 0.025, 'Frequency (%)', 'Original simulation frequency of')

## Sample simulation

from datetime import datetime start\_time = datetime.now()

np.random.seed(123)

eff\_sharp\_samp = np.zeros([1000,3]) eff\_max\_samp = np.zeros([1000,3]) satis\_samp = np.zeros([1000,3]) naive\_samp = np.zeros([1000,3])

for i in range(1000): if i == 0:

sharp\_weights = eff\_sharp\_wts max\_weights = eff\_max\_wts sat\_weights = satis\_wts nav\_weights = simple\_wts

else:

samp1 = int(np.random.choice(1000,1))

\_, sharp\_wts = max\_sharpe(sim1[samp1][0])

\_, max\_wts = max\_ret(sim1[samp1][0])

sharp\_weights = sharp\_wts max\_weights = max\_wts

test = port\_sim\_wts(sim1[samp1][0], 1000, 4, 0.07, 0.1) if np.isnan(test.any()):

sat\_wts = satis\_wts else:

sat\_wts = test

samp2 = int(np.random.choice(1000,1))

eff\_sharp\_samp[i,:2] = port\_func(sim1[samp2][0], sharp\_wts) eff\_max\_samp[i,:2] = port\_func(sim1[samp2][0], max\_wts) satis\_samp[i,:2] = port\_func(sim1[samp2][0], sat\_wts) naive\_samp[i,:2] = port\_func(sim1[samp2][0], nav\_wts)

eff\_sharp\_samp[:,2] = eff\_sharp\_samp[:,0]/eff\_sharp\_samp[:,1] eff\_max\_samp[:,2] = eff\_max\_samp[:,0]/eff\_max\_samp[:,1] satis\_samp[:,2] = satis\_samp[:,0]/satis\_samp[:,1] naive\_samp[:,2] = naive\_samp[:,0]/naive\_samp[:,1]

end\_time = datetime.now()

print('Duration: {}'.format(end\_time - start\_time)) # Duration: 0:07:19.733893

# Create sample list and graph

samp\_list = [eff\_sharp\_samp, eff\_max\_samp, satis\_samp, naive\_samp]

port\_means\_samp = [] for df in samp\_list:

port\_means\_samp.append(np.mean(df[:,0])\*1200)

# Sample graph

pf\_graf(port\_names, port\_means\_samp, 1, 0.5, 'Returns (%)', 'Random sample simulation return')

# Original graph

pf\_graf(port\_names, port\_means1, 1, 0.5, 'Returns (%)', 'Original simulation return')

# Calculate comparison values for sample simulation comp\_values\_samp = []

for i in range(4):

for j in range(i+1, 4):

comps\_samp = np.mean(samp\_list[i][:,0] > samp\_list[j][:,0]) comp\_values\_samp.append(comps\_samp)

# Sample graph

pf\_graf(comp\_names[:-1], comp\_values\_samp[:-1], 2, 0.025, 'Frequency (%)', 'Random sample simulation frequency of')

# original graph

pf\_graf(comp\_names[:-1], comp\_values1[:-1], 2, 0.025, 'Frequency (%)', 'Original simulation frequency of')