The project is structured into three parts, risk, alpha, and optimization.

To begin with, risk is measured by monthly covariance and variance matrix. We select monthly active stocks to calculate the past years' covariance and variance matrix and shrinks the variance towards its target.

In terms of alpha, this project adapts short-term mean-reversion of returns, short-term momentum of returns,
long-term value of market-to-book ratio, and long-term momentum of returns. To remove outliers and extreme values. we demean, standardize and winsorize every day each of these four alphas individually before blending them up with the specified weight

Once we prepare our monthly risk and alpha, we proceed to optimization. We use the quartic optimization which deducts trading cost and risk from alpha returns. The constraint is weight of each stock is between 0 to 1.

In [ ]:
```
import pandas as pd
import numpy as np
import datetime

import pickle

# pickle.dump(alldata, outfile)
# outfile.close()
# import the file "alldata" to check if it worked
filename = 'alldata_sol'
infile = open(filename,'rb')
new_file = pickle.load(infile)
infile.close()
print(new_file.keys())
```

In [175]:
```
industry_list =  new_file['industrylist']['industry'].unique()
industry_df = new_file['industrylist']
```

In [176]:
```
## assign industry dummy, dimensions n * rho. n is the number of stock and rho is the number of industry

for industry in industry_list:
    industry_df[industry] = 0
    industry_df[industry] = np.where(industry_df['industry'] == industry, 1 , 0)

industry_df = industry_df.drop(columns=['industry','date','dscode'])

print(industry_df)
print(industry_df.shape)
```

```
     FNSVS  PERSG  INDMT  GNRET  TRLES  INDTR  TELFL  MEDIA  OILES  CHMCL  \
0        1      0      0      0      0      0      0      0      0      0   
1        0      1      0      0      0      0      0      0      0      0   
2        0      0      1      0      0      0      0      0      0      0   
3        0      0      0      1      0      0      0      0      0      0   
4        0      0      0      0      1      0      0      0      0      0   
..     ...    ...    ...    ...    ...    ...    ...    ...    ...    ...   
561      0      0      0      0      0      0      0      0      0      0   
562      0      0      0      0      0      0      0      0      0      0   
563      0      0      0      0      1      0      0      0      0      0   
564      0      0      0      0      0      0      0      1      0      0   
565      0      0      0      0      0      0      0      0      0      0   

     ...  BEVES  LEISG  FSTPA  RLISV  EQINV  HHOLD  FDRGR  GNIND  MNING  REITS  
0    ...      0      0      0      0      0      0      0      0      0      0  
1    ...      0      0      0      0      0      0      0      0      0      0  
2    ...      0      0      0      0      0      0      0      0      0      0  
3    ...      0      0      0      0      0      0      0      0      0      0  
4    ...      0      0      0      0      0      0      0      0      0      0  
..   ...    ...    ...    ...    ...    ...    ...    ...    ...    ...    ...  
561  ...      0      0      0      0      0      0      0      0      0      0  
562  ...      0      0      0      0      0      0      0      0      0      0  
563  ...      0      0      0      0      0      0      0      0      0      0  
564  ...      0      0      0      0      0      0      0      0      0      0  
565  ...      0      0      0      0      0      0      0      0      0      0  

[566 rows x 40 columns]
(566, 40)
```

Risk

In [177]:
```
f = 'database.pkl'
i = open(f,'rb')
nf = pickle.load(i)
i.close()
nf.keys()
```

Out[177]:
```
dict_keys(['myday', 'price', 'tri', 'volume', 'mtbv', 'rec', 'isactivenow', 'tcost', 'cap'])
```

In [178]:
```
# get the index of first day of each month from 1998 - 2002

myday = pd.DataFrame(pd.to_datetime(nf['myday']))
myday['year'] = myday['V1'].dt.year
myday['month'] = myday['V1'].dt.month

first_day_index = []
for i in range(1998,2003):
    for j in range(1,13):
        year_data = myday.loc[myday['year'] == i]
        monthly_data =  year_data.loc[year_data['month'] == j]
        idx = int(monthly_data[monthly_data['year']== i].index[0])
        first_day_index.append(idx)
```

In [179]:
```
# index of first day of the month
d = [print(x, end=', ') for x in first_day_index]
```

```
245, 265, 285, 307, 326, 344, 365, 386, 407, 429, 451, 471, 491, 511, 531, 554, 574, 593, 615, 637, 659, 681, 702, 724, 745, 766, 787, 810, 828, 850, 871, 89
2, 915, 936, 958, 980, 999, 1021, 1041, 1063, 1082, 1104, 1124, 1146, 1169, 1189, 1212, 1234, 1251, 1273, 1293, 1313, 1334, 1356, 1376, 1399, 1421, 1442, 146
5, 1486,
```

In [180]:

```
# get the monthly active securities from 1998 - 2002

monthly_active_dic = {}

for i in first_day_index:
    is_active = nf['isactivenow'].iloc[i,:]

    # get the monthly active securities
    active_list = []
    for j in range(len(is_active)):
        if is_active[j]== 1 :
            active_list.append(j)

    monthly_active_dic[i] = active_list
```

In [181]:

```
# example of monthly active securities: the active securitie of 245th row which is Jan 1998
m = [print(monthly_active_dic[245])]
```

```
[1, 8, 9, 14, 17, 20, 21, 23, 26, 29, 31, 108, 110, 113, 115, 116, 117, 118, 119, 125, 132, 133, 138, 140, 144, 146, 147, 150, 151, 223, 224, 229, 234, 240, 2
41, 242, 244, 246, 247, 248, 249, 250, 259, 262, 264, 266, 267, 268, 269, 270, 272, 274, 291, 293, 295, 296, 299, 301, 305, 306, 308, 309, 318, 320, 342, 351,
355, 356, 357, 358, 359, 360, 362, 363, 364, 366, 367, 368, 370, 371, 373, 374, 377, 379, 380, 382, 383, 384, 385, 388, 389, 393, 395, 396, 397, 399, 400, 40
1, 402, 403, 407, 408, 409, 410, 411, 413, 414, 416, 418, 419, 441, 443, 444, 445, 446, 447, 449, 452, 454, 458, 459, 460, 461, 464, 465, 466, 470, 471, 472,
473, 474, 475, 480, 482, 483, 485, 489, 491, 492, 493, 496, 497, 498, 499, 500, 501, 502, 509, 512, 513, 514, 516, 517, 518, 520, 521, 522, 524, 526, 527, 53
6, 537, 543, 544, 545, 547, 548, 549, 551, 552, 558, 559, 560, 562, 564]
```

In [182]:

```
tri_df = pd.concat([nf['tri'], myday],axis = 1)
# backfill NaN with the first value
tri_df = tri_df.bfill(axis ='rows')
```

In [183]:

```
shrink = []
count = 0
shrinkage_covs = {}

## Covariance matrix is adjusted every month and included active stocks only.
## S is sample covariance without demean

for i in first_day_index:

    # get current year and current data
    curr_data = tri_df.iloc[i,:]
    curr_year = tri_df.iloc[i,:]['year']

    # get previous year and previous data
    pre_year = curr_year - 1
    pre_data = tri_df.loc[tri_df['year'] == pre_year]

    # get previous active securities for the past year
    pre_act_data = pre_data.iloc[:, monthly_active_dic[i]]
#     pre_act_data_mean = pre_data.iloc[:, monthly_active_dic[i]].mean()

    # n is the numbes of stock and T is the numbes of days
    n = pre_act_data.shape[1]
    T = pre_act_data.shape[0]

    # calculate sample covariance without demean. S = 1/T (X X') = 1/ T sum_from_t_to_T (Xt Xt')
    S = np.dot(np.transpose(np.array(pre_act_data)), np.array(pre_act_data))
    S = S / T

    # calculate estimation error. omega square = 1/(T * (T-1)) sum_from t = 1 to T ||Xt Xt' - S||** 2
    omega_square = 0

    for j in range(len(pre_act_data)):
        squared_X = np.dot(np.transpose(np.array([pre_act_data.iloc[j,:]])), np.array([pre_act_data.iloc[j,:]]))
        distance  = (squared_X - S)
        omega_square += ((distance **2).sum())

    omega_square = omega_square / (T * (T-1))

    # calculate shrinkage target. target = average of variance
    shrinkage_target = (pre_act_data.var()).sum()/n

    # calculate dispersion. delta square = (norm of (S - shrinkage * I)) ** 2 - omega **2
    delta_square = ((S - shrinkage_target * np.identity(n))**2).sum() - omega_square

    shrinkage_intensity =  delta_square / (delta_square + omega_square)

    shrink.append(shrinkage_intensity)

    shrinkage_covs[i] = (1 - shrinkage_intensity) * np.dot(shrinkage_target, np.identity(n))+\
                        shrinkage_intensity * S
```

In [184]:

```
s = [print(x, end=', ') for x in shrink]
```

```
0.9998703235573941, 0.9998694663896874, 0.9998688044528415, 0.9998694290251778, 0.9998685907376671, 0.9998688884521405, 0.9998688388637115, 0.999868873069229
7, 0.9998689360781063, 0.9998697799788795, 0.9998696629752982, 0.9998697106556815, 0.9997709604958691, 0.9997713185079514, 0.999771870288715, 0.99977193331171
07, 0.9997719439335386, 0.9997715237298743, 0.9997720440814417, 0.9997721001479238, 0.9997720347730847, 0.9997721221650553, 0.9997720358720618, 0.999771828744
8497, 0.9996473451365823, 0.999647601487138, 0.9996498328592794, 0.9996464400306613, 0.9996780872772286, 0.9998557895396879, 0.9998558483960899, 0.99967622661
98813, 0.9996795601876429, 0.9998553203770386, 0.999855330082757, 0.9998553809017454, 0.9998000860627724, 0.9997774825391427, 0.9998001209117172, 0.9998001842
206694, 0.9997972617203499, 0.9997978991309118, 0.9997966108218581, 0.9997987720938323, 0.9997977690417075, 0.9997983826041816, 0.9997979730088684, 0.99979798
82307911, 0.9997460532100103, 0.9997460107029177, 0.9997486595668734, 0.9997468334408459, 0.9997468089727465, 0.9997470566284986, 0.999748112236281, 0.9997498
626839456, 0.999751233111221, 0.9997520330571875, 0.9997514878697095, 0.999749756370738,
```

Alpha

In [185]:

```python
# demean, standardize and winsorize data
# input is dataframe and output is np.array

def demean_standardize_win(df):
    for i in range(len(df)):

        #calculate cross-industry mean on date i
        mean = df.iloc[i,:].mean()

        #calculate cross industry standard deviation on date i
        std = df.iloc[i,:].std()

        #standardlize ret by substracting cross-industry mean and then divided by standard deviation
        df.iloc[i,:] = (df.iloc[i,:] - mean)/std

    #winsorize data. If z score > 3, bring it down to 3. If z score < -3, bring it up to -3.
    df = np.where(df> 3, 3, df)
    df = np.where(df < -3, -3, df)

    return df
```

a

In [186]:

```python
import numpy as np
w1 = 0.50

#arithmetic mean of past 21 days return
ari_ret = nf['tri'].rolling(21).mean()

#backfill NAN
# ari_ret = ari_ret.bfill(axis ='rows')
# fill nan with 0
ari_ret = ari_ret.fillna(0)

#calculate industry variable
industry_var = np.identity(len(industry_df)) - np.dot(np.dot(industry_df, \
                np.linalg.inv(np.dot(industry_df.T, industry_df))),industry_df.T)

# dot product of arithmetic mean and industry variable
alpha_t = np.dot(-ari_ret, industry_var)

alpharev = demean_standardize_win(pd.DataFrame(alpha_t))
```

In [187]:

```python
print(alpharev)
print(alpharev.shape)
```

```
[[        nan        nan        nan ...        nan        nan        nan]
 [        nan        nan        nan ...        nan        nan        nan]
 [        nan        nan        nan ...        nan        nan        nan]
 ...
 [0.26574971 0.52218755 0.03513491 ... 0.16966129 0.36950031 0.27013634]
 [0.26628602 0.52395278 0.03534247 ... 0.16867638 0.36927013 0.26973781]
 [0.26678904 0.52553136 0.0354849  ... 0.16769193 0.36885335 0.26939215]]
(1504, 566)
```

b

In [188]:

```python
w2 = 0.25

#arithmetic mean of past 45 days return
alpharec = nf['rec'].rolling(45).mean()

# see function for more details
alpharec = demean_standardize_win(alpharec)

#fill NAN with 0
# alpharec = pd.DataFrame(alpharec).bfill(axis='row')
alpharec = pd.DataFrame(alpharec).fillna(0)
```

In [189]:

```python
print(alpharec.tail())
print(alpharec.shape)
```

```
            0         1         2         3         4         5         6   \
1499 -0.366364 -0.366364 -0.366364 -1.917701  0.150748  0.667861 -0.883476
1500 -0.364470 -0.364470 -0.364470 -1.923336  0.155152  0.674774 -0.884092
1501 -0.374842 -0.374842 -0.374842 -1.430368  0.152921  0.680684 -0.902605
1502 -0.382192  0.148006 -0.382192 -1.442587  0.148006  0.678203 -0.912389
1503  0.117224  0.117224 -0.413568 -1.475152  0.117224  0.648016 -0.413568

            7         8         9    ...       556       557       558  \
1499  0.667861  0.667861 -0.366364  ...  1.184973 -0.366364 -2.951925
1500  0.674774  0.674774 -0.364470  ...  0.674774 -0.364470 -2.962579
1501  0.680684  0.680684 -0.374842  ...  0.680684 -0.374842 -3.000000
1502  0.678203  0.678203 -0.382192  ...  0.678203 -0.382192 -3.000000
1503  0.648016  1.178808 -0.413568  ...  0.648016 -0.413568 -3.000000

           559  560       561       562       563       564       565
1499  0.150748 -3.0 -2.434813 -0.366364  0.667861  0.667861  0.150748
1500 -0.364470 -3.0 -2.442957 -0.364470  0.674774  0.674774  0.155152
1501 -0.374842 -3.0 -1.958131 -0.374842  0.680684  0.680684  0.152921
1502 -0.382192 -3.0 -1.972784 -0.382192  0.678203  0.678203  0.148006
1503  0.117224 -3.0 -2.005944 -0.413568  0.648016  0.117224  0.117224

[5 rows x 566 columns]
(1504, 566)
```

c

In [190]:
```
w3 = 0.15
alphaval = nf['mtbv']
alphaval = demean_standardize_win(alphaval)

# alphaval = pd.DataFrame(alphaval).bfill(axis='row')
alphaval = pd.DataFrame(alphaval).fillna(0)
```

In [191]:
```
print(alphaval.tail())
print(alphaval.shape)
```

```
            0         1         2         3         4         5         6    \
1499 -0.096189 -0.012860 -0.441408  2.034646 -0.020796 -0.377920 -0.258879
1500 -0.055241 -0.027866 -0.446325  2.083985 -0.031776 -0.372019 -0.258605
1501 -0.028607 -0.048197 -0.443901  1.996932 -0.036443 -0.377298 -0.263679
1502 -0.027222 -0.046805 -0.434537  2.091600 -0.035055 -0.360124 -0.250462
1503  0.008576 -0.054471 -0.452457  2.116721 -0.042650 -0.350005 -0.243613

            7         8         9    ...  556       557       558       559  \
1499 -0.366016  0.407751  0.074436  ...  0.0 -0.532673 -0.028732 -0.568385
1500 -0.360287  0.425791  0.046440  ...  0.0 -0.532363 -0.047420 -0.575382
1501 -0.365544  0.402359  0.053668  ...  0.0 -0.537930 -0.040361 -0.573191
1502 -0.352291  0.391842  0.086356  ...  0.0 -0.520700 -0.046805 -0.567698
1503 -0.350005  0.390801  0.075564  ...  0.0 -0.503683 -0.046590 -0.578552

            560       561       562       563       564       565
1499 -0.231102 -0.147774 -0.191422 -0.564417 -0.183486 -0.568385
1500 -0.246872 -0.145191 -0.196031 -0.563650 -0.188210 -0.583204
1501 -0.248008 -0.134390 -0.189240 -0.561438 -0.181404 -0.573191
1502 -0.250462 -0.148633 -0.183882 -0.555949 -0.179965 -0.567698
1503 -0.239672 -0.172685 -0.172685 -0.550969 -0.176625 -0.574611

[5 rows x 566 columns]
(1504, 566)
```

d

In [192]:
```
w4 = 0.1
#drop unnecessary columns
# updated_df = nf['tri'].drop(columns=['Year','date'])
updated_df = nf['tri']
#cumulative sum of last 11 months
alphamom = updated_df.rolling(11).sum()
alphamom = demean_standardize_win(alphamom)
alphamom  = pd.DataFrame(alphamom ).fillna(0)
```

In [193]:
```
print(alphamom.tail())
print(alphamom.shape)
```

```
            0         1         2         3         4         5         6    \
1499 -0.400469 -0.294335 -0.394852 -0.384865 -0.365408 -0.423679 -0.433366
1500 -0.400213 -0.294329 -0.394615 -0.384751 -0.365039 -0.423507 -0.432975
1501 -0.399822 -0.294239 -0.394339 -0.384453 -0.364547 -0.423289 -0.432565
1502 -0.399511 -0.294590 -0.394105 -0.384413 -0.364326 -0.423213 -0.432278
1503 -0.399098 -0.293832 -0.393755 -0.384012 -0.363708 -0.422909 -0.431827

            7         8         9    ...       556       557       558  \
1499 -0.436247 -0.401768 -0.368661  ... -0.112233 -0.308763 -0.386826
1500 -0.435883 -0.401285 -0.368201  ... -0.116650 -0.304405 -0.386811
1501 -0.435480 -0.400818 -0.367659  ... -0.120487 -0.299917 -0.386715
1502 -0.435173 -0.400377 -0.367373  ... -0.123889 -0.295641 -0.386765
1503 -0.434706 -0.399855 -0.366944  ... -0.126829 -0.291658 -0.386551

            559       560       561       562       563       564       565
1499 -0.437824  0.808725 -0.303641 -0.193947 -0.440468 -0.347266 -0.423323
1500 -0.437582  0.805217 -0.303626 -0.193317 -0.440132 -0.347424 -0.422563
1501 -0.437332  0.801158 -0.303568 -0.192950 -0.439758 -0.347634 -0.421710
1502 -0.437161  0.795293 -0.303789 -0.192717 -0.439487 -0.348087 -0.420912
1503 -0.436811  0.791636 -0.303898 -0.191844 -0.439038 -0.348339 -0.419866

[5 rows x 566 columns]
(1504, 566)
```

In [194]:
```
alphablend = w1 * alpharev + w2 * alpharec + w3 * alphaval + w4 * alphamom
alphablend = demean_standardize_win(alphablend)
```

In [195]:
```
print(alphablend)
print(alphablend.shape)
```

```
[[         nan         nan         nan ...         nan         nan
          nan]
 [         nan         nan         nan ...         nan         nan
          nan]
 [         nan         nan         nan ...         nan         nan
          nan]
 ...
 [-0.04622059  0.28502074 -0.47796275 ...  0.2754625   0.6805854
   0.07633201]
 [-0.04954239  0.60759782 -0.47966229 ...  0.27540805  0.68061897
   0.07523037]
 [ 0.26622586  0.58347651 -0.50469323 ...  0.25460922  0.33512315
   0.05158075]]
(1504, 566)
```

Optimization

In [196]:
```
import cvxopt
from cvxopt import matrix, solvers

## showing the result instead of progress
# solvers.options['show_progress'] = False
# solvers.options['feastol'] = 1e-20
```

In [197]:

```python
# total weights contain active stocks only. Create weights that have both active stocks and inactive stocks
# initiate a array with 0, lengh is equal to total securities

# loop through 60 months to get first day index
def fill_active_inactive_stocks(t, matrix):
    n = 566
    x = np.array(np.zeros(n))

    # find the index position of active stocks using index stored in monthly active dic
    pos = monthly_active_dic[t]

    for i in range(0,len(pos)):
        x[pos[i]] = matrix[i]

    return x
```

In [ ]:

```python
mu = 10
lam = 5
n = 566
daily_trading_constraint = 15
start = 245
T = 1504

# m controls monthly dynamic variables
m = 0
total_weights = []
for t in range(start, T):
    # get previous day row index
    t_minus_1 = t - 1

    # control monthly dynamic variables which are covariance matrix, numbers of active stocks,
    # initial weight, and monthly active stocks
    if m < len(first_day_index) and t == first_day_index[m]:

        # get monthly shrinkage covariance
        risk = matrix(shrinkage_covs[t],tc ='d')

        # get monthly numbers of active stocks
        numbers_of_active_stocks = len(monthly_active_dic[t])

        # The CXVOPT solver only accepts matrices containing doubles,
        # and if a list containing only integers was supplied to the matrix constructor,
        # it will create an integer matrix and eventual lead to a cryptic error.
        # dimension of initial weight is numbers of active stock * 1
        # initial weight is 0 for every month
        w = matrix(np.zeros(numbers_of_active_stocks), tc='d')

        monthly_active_stocks = monthly_active_dic[t]

        # switch to next month
        m+=1

    # active stocks' alpha performance on day t -1
    alpha = matrix(np.array([alphablend[t_minus_1][x] for x in monthly_active_stocks]),tc = 'd')

    # active stocks' cost which includes bid/offer spread cost and commission cost on the execution day, day t-1
    tau = matrix(np.array([nf['tcost'].iloc[t_minus_1,:][x] for x in monthly_active_stocks]),tc='d')

    # solver's param, H
    H = 2 * mu * matrix([[risk, - risk],[-risk,risk]],tc ='d')

    # solver's param, g
    g1 = np.array(2* mu * np.dot(risk, w) - alpha + lam * tau)
    g2 = np.array(-2* mu * np.dot(risk, w) + alpha + lam * tau)

    # fill nan with the closet value
    mask1 = np.isnan(g1)
    g1[mask1] = np.interp(np.flatnonzero(mask1), np.flatnonzero(~mask1), g1[~mask1])
    mask2 = np.isnan(g2)
    g2[mask2] = np.interp(np.flatnonzero(mask2), np.flatnonzero(~mask2), g2[~mask2])

    g = matrix(np.vstack([g1, g2]), tc ='d')


    # inequality constraint (Gx < h): 0 <= y, z <= 1, weight of singal equity is between 0 and 1
    identity = matrix(np.identity(numbers_of_active_stocks), tc = 'd')
    UB = matrix(np.array([2] * (numbers_of_active_stocks)), tc = 'd')
    LB = matrix(np.zeros(numbers_of_active_stocks),tc= 'd')
    A = matrix([[identity, - identity],[identity, - identity]])
    b = matrix([UB, LB])

    # eqality constraint (Ax =b): daily trading volume is less than 15 m, long position and short position is equal
#     c1 = np.array([mu] * (numbers_of_active_stocks*2))
#     c1 = np.array([[mu] * (numbers_of_active_stocks*2)])
#     c2 = np.array([np.ones(numbers_of_active_stocks)])
#     c3 = np.array([np.zeros(numbers_of_active_stocks)])
#     print(c1.shape, c2.shape, c3.shape)

#     # C = matrix(np.vstack((np.hstack((c2, c3)), np.hstack((c3, c2)))), tc = 'd')
#     C = matrix(np.vstack((c1,np.hstack((c2, c3)), np.hstack((c3, c2)))), tc = 'd')
#     print(C.size)
#     d = matrix(np.array([daily_trading_constraint*2,0 ,0]), tc = 'd')
#     # d = matrix(np.array([0 ,0]), tc = 'd')
# #     sol = solvers.qp(H,g, A, b)

    sol = solvers.qp(H,g, A, b)
    y = sol['x'][:numbers_of_active_stocks]
    z = sol['x'][numbers_of_active_stocks:]

    # calculate final weight x, x = w + y - z
    final_weights = w + y - z

    #today's weights is initial weights of the next trading day
    w = final_weights

    # np.squeeze: show result instead of data type
    # w contain active stocks only. Create weights that have both active stocks and inactive stocks
    x = fill_active_inactive_stocks(first_day_index[m-1], np.squeeze(w))
    total_weights.append(x)
```

In [173]:

```
n = 566
T = 1504
trade = np.zeros((T -len(total_weights), n))
print(trade)
print(trade.shape)
trade = np.vstack((trade, np.array(total_weights)))
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
(245, 566)
```

In [167]:

```
trade.shape
```

Out[167]:

```
(1504, 566)
```

In [ ]:

```

```