FINA 4380: Cointegration-based pair trading with DCC-eGARCH volatility clustering and portfolio weight optimization

LAM Lok* CHEUNG Chun Hin† NG Shing Cheong‡ SIN Chi Man§

May 3, 2024

1 Introduction

In our trading strategy, we employ cointegration-based pair trading, capitalizing on deviations from the absolute price spreads of stock pairs that exhibit long-term equilibrium relationships. This approach is enhanced by the integration of DCC-eGARCH models, which dynamically forecast volatility and calculate z-scores to generate trading signals. We further refine our strategy through an initial equal weighting of each pair, followed by optimization of portfolio weights where the amount of investment in each pair is determined by its confidence score. **(Github link)**

2 Pair Formation[1]

2.1 Asset pool

We selected a diversified portfolio of approximately 50 US equities and ETF, spanning various sectors and categories including commodities, technology, healthcare, and banking. which contains large market capitalization with high liquidity.

GDX	**GDXJ**	**GLD**	**AAPL**	**GOOGL**
META	**AMD**	**NVDA**	**CSCO**	**ORCL**
TTWO	**EA**	**HYG**	**LQD**	**JNK**
SLV	**USLV**	**SIVR**	**USO**	**UWT**
QQQ	**SPY**	**VOO**	**VDE**	**VTI**
EMLP	**VDC**	**FSTA**	**KXI**	**IBB**
VHT	**VNQ**	**IYR**	**MSFT**	**PG**
TMF	**UPRO**	**WFC**	**JPM**	**GS**
CVX	**XOM**	**INTC**	**COST**	**WMT**
T	**VZ**	**CMCSA**	**AMZN**	

Table 1: Selected Stocks and ETF to form cointegrated pair

*1155176058
†11551176617
‡1155174438
§1155159539

[1]Please refer to the code: `Pairformation_HedgeRatio.py`

The dataset was segmented into training data which comprising 80% of the total spanning from January 2012 to August 2021, and testing data which constituting the remaining 20% and covering the period from August 2021 to December 2023.

2.2 Cointegration[2]

To identify potential cointegrated pairs from the portfolio, assets x_t and y_t is determined to be conintegrated if the absolute price spread $z_t = y_t - (\beta x_t + \alpha)$ is stationary, which means z_t has constant mean and auto-covariance. We use python function `coint` (which uses the Engle-Granger test) to determine the stationarity of the spread.

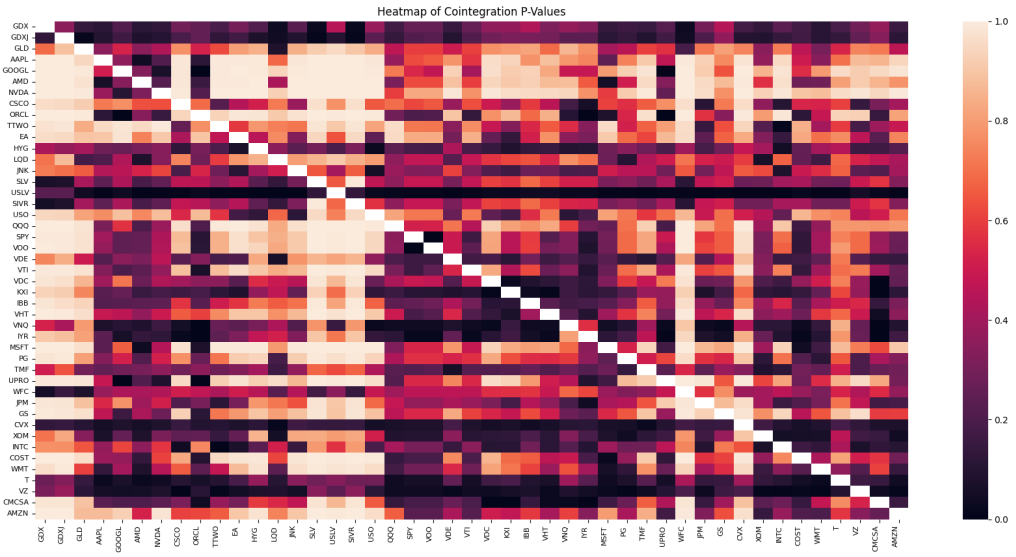


Figure 1: Heatmap of P-value for the formation of cointegrated pairs

If P-value for the hypothesis test is less than critical value = 0.05, we can conclude that two assets are cointegrated to form a pair trade. We formed 73 cointegrated pairs according to this rule.

2.3 Kalman Filter and Half Life[3]

Due to general factors including changing market condition or volatility fluctuation, hedge ratio is not always constant. Therefore, we implement the function `KalmanFilterRegression(x,y)`, which is a Kalman Filter method to estimate state variables dynamic hedge ratio β_t to determine how many units of asset `(x)` to hedge against `(y)` during the testing period.

Before running the function of `KalmanFilterRegression(x,y)`, we use the function `KalmanFilterAverage(x)` to smooth the price series of two asset `(x,y)` in each pair by obtaining their rolling mean. This helps us filter out the price noise of individual asset, and implement a better signal clarity when we execute trades.

The function `half_life(spread)` calculates the half-life of the spread between two assets in each pair trades, which is a measure of how quickly the spread reverts to its mean. This helps determine the days of rolling window for mean, volatility and z-score of spread, which are crucial for generating trading signal in the later part.

[2]Please refer to the code: `Pairformation_HedgeRatio.py`
[3]Please refer to the code: `Pairformation_HedgeRatio.py`

3 Trading Signal

We propose a z-score-based trading strategy that quantifies how far the current spread deviates from the mean in terms of standard deviations. This approach leverages spreads' mean-reverting property to generate systematic and rule-based signals for entering and exiting positions. The Z-score is defined as $Z = \frac{X-\mu}{\sigma}$, where X represents the spread, μ is the mean of the spread, and σ represents the standard deviation of the spread.

The trading signals are defined by four parameters: long entry threshold (LET), long exit threshold (LETX), short entry threshold (SET), and short exit threshold (SETX), with the following trading conditions:

- For a long entry, the z-score falls below the long entry threshold, indicated by $z - score_t < \text{LET} < z - score_{t-1}$,
- To exit a long position, the z-score rises above the long exit threshold, indicated by $z - score_{t-1} < \text{LETX} < z - score_t$,
- For a short entry, the z-score rises above the short entry threshold, indicated by $z - score_{t-1} < \text{SET} < z - score_t$.
- To exit a short position, the z-score falls below the short exit threshold, indicated by $z - score_t < \text{SETX} < z - score_{t-1}$.

In the following section, we will employ the DCC-EGARCH model to estimate volatility and perform a grid search to optimize the trading signal parameters.

3.1 Volatility Clustering

First, we assessed the normality of the spread between pairs using the Shapiro-Wilk (SW) test and the Jarque-Bera (JB) test. The SW test was utilized to measure the departure of the spread from normality, while the JB test compared the skewness and kurtosis of the spread with that of a normal distribution.
Upon analyzing the training dataset, none of the pairs were likely to be normally distributed under the 0.05 significance level for both the SW and JB tests. Moving on to the testing dataset, we found that only a small portion of the pairs exhibited indications of normal distribution. Specifically, based on the SW test, barely 10 pairs out of the total were likely to be normally distributed, while the JB test suggested that 18 pairs showed signs of normality. These results highlight a limited presence of normality in the spread between pairs.

To gain further insights, additional analysis was conducted using quantile-quantile (Q-Q) plots. The examination of these plots revealed the presence of heavy-tailed distributions in the spread. Consequently, we have chosen to adopt the studentized-t distribution, which accounts for heavy-tailed data.

Engle (1982) initially proposed auto-regressive conditional heteroscedastic (ARCH) models for estimating time series. Since then, various types of ARCH models with different specifications have emerged in the literature, including the GARCH, TGARCH, EGARCH, and PGARCH models. The GARCH model depicts the history of shocks in a series with a changing conditional variance, while the TGARCH and EGARCH models accommodate asymmetric shock to volatility.
Nelson calibrated the EGARCH (Exponential Generalized Autoregressive Conditional Heteroskedasticity) model in 1991. It captures the empirical observations that negative shocks at $time_{t-1}$ have a stronger impact on the variance at $time_t$ than positive shocks, a phenomenon known as the leverage effect. Another advantage of the EGARCH model is that the variance will be positive even if the parameters are negative.

$$\epsilon_t \sim EGARCH(p,q) \quad \text{if} \quad \epsilon_t = \sigma_t z_t, \quad \text{where} \quad z_t \sim \text{studentized-t distribution} \tag{1}$$

$$\ln(\sigma_t^2) = \omega + \sum_{i=1}^{p} (\alpha_i |z_{t-i} - E(z_{t-i})| + \gamma_i z_{t-i}) + \sum_{j=1}^{q} \beta_j \ln(\sigma_{t-j}^2) \tag{2}$$

where:

- $\ln(\sigma_t^2)$ is the natural logarithm of the conditional variance,
- ω is a constant, γ_i measures the asymmetric effect due to leverage,
- α_i and β_j measures the ARCH and GARCH effects respectively.

In the model selection process, we performed a grid search on the hyperparameters (p) and (q) to determine the optimal values. We systematically tested combinations of p ranging from 1 to 5 and q ranging from 1 to 5. To assess the performance of each combination, we computed the Akaike information criterion (AIC) and Bayesian information criterion (BIC).
Although the results showed that the lowest IC values occurred when p=3 and q=1 (Figure 1), the IC values at p=q=1 are sufficiently low. For the sake of simplicity, we have chosen the EGARCH (1,1) model for our further analysis since it demonstrates a promising fit based on both the mean and median values of AIC and BIC.

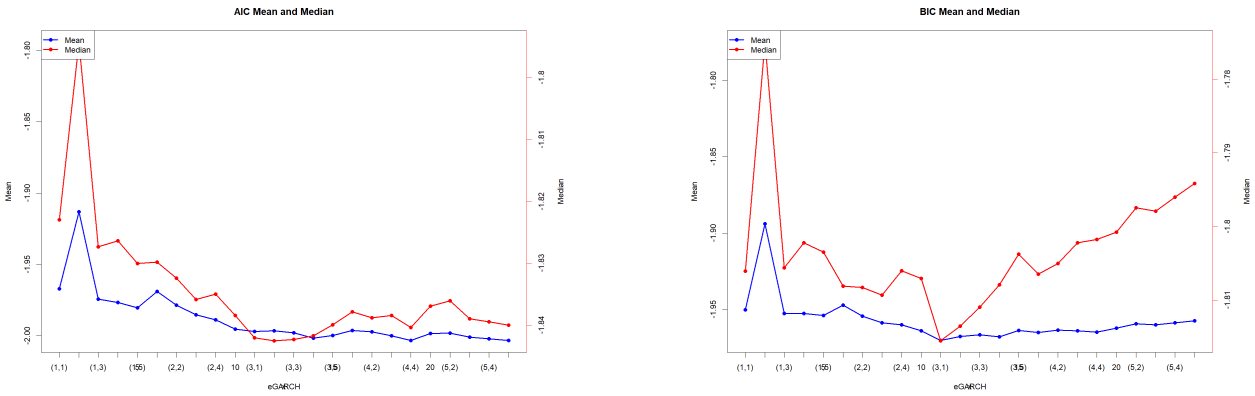


Figure 2: Left - Akaike information criterion (AIC) , Right- Bayesian information criterion (BIC)

Next, we analyze volatility clustering in a multivariate context using DCC-EGARCH (Dynamic Conditional Correlation-EGARCH) to estimate the conditional covariance between multiple spreads using R.

$$v_t = D_t^{-1}(r_t - \mu), \quad D_t = \text{diag}\left(\sqrt{\sigma_{1t}}, \sqrt{\sigma_{2t}}, \ldots, \sqrt{\sigma_{mt}}\right), \quad \bar{Q} = \frac{1}{T} \sum_{t=(T/3)}^{T} v_{t-1} v_{t-1}^T \tag{3}$$

$$Q_t = (1 - \theta_1 - \theta_2)\bar{Q} + \theta_1 v_{t-1} v_{t-1}^T + \theta_2 Q_{t-1} \tag{4}$$

where:

- r_t is the vector of spread at time t, μ_t is the vector of conditional means,
- D_t is the diagonal matrix of conditional standard deviations,
- v_t is the standardized spread,
- Q_t is the conditional covariance matrix of the standardized spread,
- θ_1 and θ_2 are parameters controlling the persistence of volatility.

```
train.spread_scaled = scale(na.omit(train.pair_spread), center = TRUE, scale =
    TRUE)
train.length = nrow(train.spread_scaled); train.length
garch.spec = ugarchspec(variance.model = list(model="eGARCH",
    garchOrder=c(1,1)), mean.model =
    list(armaOrder=c(1,1),include.mean=FALSE),distribution.model = "std")
```

```
dcc_spec = dccspec(uspec = multispec(replicate(garch.spec, n=pairs.no)),
    dccOrder = c(1,1), model="DCC", distribution = "mvt")
dcc_GARCH = dccfit(spec = dcc_spec, data = train.spread_scaled)
# Training Fitted covariance and volatility -----
for (day in (1:train.length)){
    train.cov_matrix[[day]] = as.matrix(as.data.frame(dcc_GARCH@mfit$Q[day]))
    train.vol[day,] = diag(train.cov_matrix[[day]])
}

# DCC_GARCH Parameters ------
theta1 = tail(dcc_GARCH@mfit$matcoef,3)[1,1]
theta2 = tail(dcc_GARCH@mfit$matcoef,2)[1,1]
latest.Q = as.data.frame(tail(dcc_GARCH@mfit$Q,1))
one.third = (train.length%/%3) # Burn-in first 1/3 data
Q_sum = as.matrix(as.data.frame(dcc_GARCH@mfit$Q[one.third]))
for (day in (one.third:train.length)){Q_sum = Q_sum +
    as.matrix(as.data.frame(train.cov_matrix[day]))}
Q_mean = Q_sum / length(one.third:train.length)

# DCC_GARCH Update for Testing Phase ------
Q.t1 = (1-theta1-theta2)Q_mean + theta2 * (latest.eta %% t(latest.eta)) +
    theta1 * latest.Q
test.cov_matrix[[1]] = as.matrix(Q.t1)
test.vol[1,] = diag(as.matrix(Q.t1))
for (row in 2:test.length){
    if (row == 2){Q.lag = as.matrix(Q.t1)}
    eta.lag = as.numeric(test.spread_scaled[row,])
    Q.next = (1-theta1-theta2)Q_mean + theta2 * (eta.lag %% t(eta.lag)) +
        theta1 * Q.lag
    test.cov_matrix[[row]] = as.matrix(Q.next)
    test.vol[row,] = diag(test.cov_matrix[[row]])
    Q.lag = Q.next
}
```

The training phase involved standardizing spreads and incorporating them into the DCC-EGARCH model with a multivariate studentized-t conditional distribution. The estimated values for θ_1 and θ_2 were determined to be 0.122909 and 0.672378, respectively. To ensure accuracy, the first one-third of the fitted covariance matrix was discarded as a burn-in period, while the remaining portion was utilized to calculate the $\bar{Q}$. The fitted volatility of spreads was then retrieved to compute the z-score.
During the testing phase, the covariance matrix was updated using equation (4), and the estimated volatility was extracted iteratively. This process enabled the determination of the z-score for the subsequent signal generation.

3.2 Signal Optimization

We employ two optimization methodologies on long entry threshold (LET), long exit threshold (LETX), short entry threshold (SET), and short exit threshold (SETX): maximizing profit and maximizing the Sharpe ratio.

To maximize profit, we start by defining an objective function that takes the parameters as inputs and returns the negative value of the profit. The parameters are then optimized using `scipy.minimize`, with each parameter bounded by [-1.1, 1.1]. To maximize the Sharpe ratio, everything remains the same except for changing the objective function to return the negative value of the Sharpe ratio.

```
def objective(params, spread, z_score, base, pair_name):
    long_entryZscore, long_exitZscore, short_entryZscore, short_exitZscore =
        params
    PnL = pd.DataFrame({f"{pair_name}(Z_score)":
        z_score,f"{pair_name}(Spread)": spread})
    PnL.index = pd.to_datetime(z_score.index)
    PnL['long entry'] = (PnL[f"{pair_name}(Z_score)"] < (long_entryZscore)) &
        (PnL[f"{pair_name}(Z_score)"].shift(1) > (long_entryZscore))
    PnL['long exit'] = (PnL[f"{pair_name}(Z_score)"] > (long_exitZscore)) &
        (PnL[f"{pair_name}(Z_score)"].shift(1) < (long_exitZscore))
    PnL['num units long'] = np.nan
    PnL.loc[PnL['long entry'],'num units long'] = 1
    PnL.loc[PnL['long exit'],'num units long'] = 0
    PnL['num units long'][0] = 0
    PnL['num units long'] = PnL['num units long'].fillna(method='pad')

    PnL['short entry'] = (PnL[f"{pair_name}(Z_score)"] > (short_entryZscore))
        & (PnL[f"{pair_name}(Z_score)"].shift(1) < (short_entryZscore))
    PnL['short exit'] = (PnL[f"{pair_name}(Z_score)"] < (short_exitZscore)) &
        (PnL[f"{pair_name}(Z_score)"].shift(1) > (short_exitZscore))
    PnL['num units short'] = np.nan
    PnL.loc[PnL['short entry'],'num units short'] = -1
    PnL.loc[PnL['short exit'],'num units short'] = 0
    PnL['num units short'][0] = 0
    PnL['num units short'] = PnL['num units short'].fillna(method='pad')

    PnL['numUnits'] = PnL['num units long'] + PnL['num units short']
    PnL['spread pct ch'] = ((spread - spread.shift(1)) / base).values
    PnL['port rets'] = PnL['spread pct ch'] * PnL['numUnits'].shift(1)
    PnL['cum rets'] = PnL['port rets'].cumsum()
    PnL['cum rets'] = PnL['cum rets'] + 1
    end_val = PnL['cum rets'].iat[-1]
    return(-end_val)

def backtest(spread, z_score, base, pair_name):
    result = minimize(objective, [-0.7, 0.05, 0.7, -0.05], args=(spread,
        z_score, base, pair_name), bounds=[(-1.1, 1.1), (-1.1, 1.1), (-1.1,
        1.1), (-1.1, 1.1)], method='Nelder-Mead')
    long_entryZscore, long_exitZscore, short_entryZscore, short_exitZscore =
        result.x
    best_return = -result.fun
    return best_return, long_entryZscore, long_exitZscore, short_entryZscore,
        short_exitZscore
```

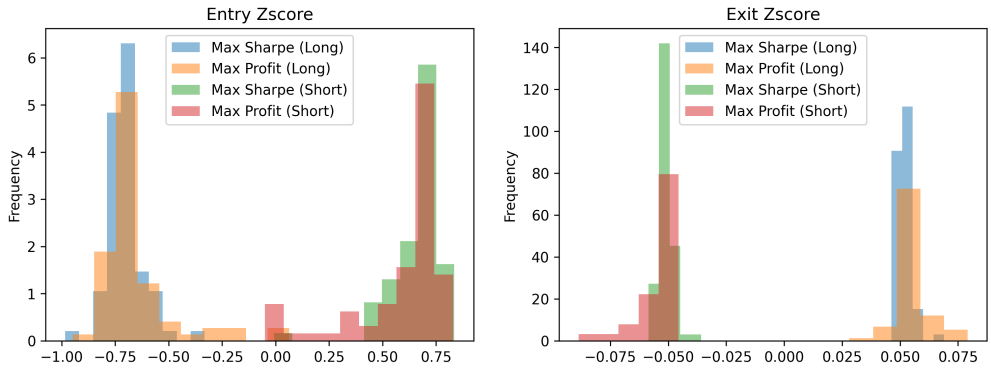


Figure 3: Left - Histogram for Entry, Right - Histogram for Exit

	MaxProfit	long_entryZscore	long_exitZscore	short_entryZscore	short_exitZscore
mean	3.437518	-0.647286	0.054060	0.587717	-0.054488
std	1.636079	0.177435	0.007117	0.220871	0.008078
min	1.212882	-0.949097	0.028142	-0.049572	-0.088670
25%	2.414950	-0.729579	0.050584	0.548860	-0.054809
50%	2.897327	-0.693437	0.051872	0.680232	-0.051939
75%	3.530215	-0.640698	0.056442	0.708750	-0.050185
max	8.043060	0.064196	0.079131	0.829552	-0.045588

	MaxSharpe	long_entryZscore	long_exitZscore	short_entryZscore	short_exitZscore
mean	2.033816	-0.700840	0.051767	0.659499	-0.050989
std	0.501662	0.087618	0.003221	0.116720	0.003282
min	0.452877	-0.983603	0.046328	-0.007080	-0.058483
25%	1.810431	-0.746912	0.049935	0.620172	-0.052893
50%	2.021001	-0.706552	0.051105	0.694531	-0.050934
75%	2.359007	-0.665103	0.053075	0.724888	-0.049531
max	3.235803	-0.332021	0.069006	0.834995	-0.035816

Figure 4: Left - Optimized Parameters for Max Profit, Right- Optimized Parameters for Max Sharpe

The optimization revealed that LET ranges from -0.65 to -0.7, LETX is approximately 0.05, SET ranges from 0.6 to 0.65, and SETX is around -0.05. For each $spread_i$, the trading strategy with the maximum profit or shape follows:

- For a long entry, the z-score falls below the long entry threshold, indicated by $z - score_{i,t} < LET_i < z - score_{i,t-1}$,
- To exit a long position, the z-score rises above the long exit threshold, indicated by $z - score_{i,t-1} < LETX_i < z - score_{i,t}$,
- For a short entry, the z-score rises above the short entry threshold, indicated by $z - score_{i,t-1} < SET_i < z - score_{i,t}$.
- To exit a short position, the z-score falls below the short exit threshold, indicated by $z - score_{i,t} < SETX_i < z - score_{i,t-1}$.

4 Portfolio weighting

We introduce a new confidence parameter γ to optimize the portfolio weight allocated to each co-integration pairs, aiming to beat an equal-weighted allocation.

4.1 Confidence Parameter (γ)

The confidence parameter γ aims to capture the historical strength and the predicted probability of short-term mean reversion of the spread.

We recall that the co-integration test is able to evaluate the significance of mean-reverting ability of two time series. Also, we notice that given a higher volatility and we enter the trade when the spread is at an extreme value, the probability of the spread going to normal value is much greater than the probability of the spread going to even extreme value. Therefore, we use the predicted volatility from DCC model as another input of the new parameter.

Incorporating all, we define the two γ as:

$$\gamma_t(i) = DCCVol_i * Coint_t(i, r) \tag{5}$$

$$\gamma_p(i) = \frac{DCCVol_i}{Coint_p(i, r)} \tag{6}$$

where:

- $DCCVol_i$ is the predicted volatility of $spread_i$,
- $Coint_t$ is the absolute value of the co-integration test statistic,
- $Coint_p$ is the co-integration test p-value,
- r is the rolling window of co-integration test,
- i is the i-th chosen co-integration pair.

4.2 Weight Calculation

We try to allocate as much weight to the pair which gives the highest confidence score while we try to limit the weight allocation to a specific pair to avoid extreme risk exposure. Specifically, we set the weight limit (around 5%) of each pair at 4 times of an equal weighted allocation. Therefore, we calculate the weight (w_i) of each pair as the following:

$$w_i = max(\frac{4}{n}, \frac{\gamma_i}{\sum_{j=1}^{n} \gamma_j}) \tag{7}$$

- γ_i is confidence score of $spread_i$,
- n is the number of chosen co-integration pairs.

To avoid there are too many pairs falling into the weight upper bound, the weight calculation algorithm[4] will use an iterative approach to prioritize weight on the most confident pairs. Whenever the upper bound is triggered by some pairs, the weight will multiply the current remaining weight to make sure $\sum_{i=1}^{n} \gamma_i = 1$.

[4]footnotePlease refer to the code: `Weight_Calculation.py`

5 Backtest Results[5]

Finally, we backtest the following strategies: Equal weight with fixed entry threshold (EW-FE), Equal weight with optimized entry threshold (EW-OE), Optimized weight with optimized entry threshold (OW-OE), using BackTrader.

5.1 BackTrader Framework

We generate the trading signals from previous output files:

```
test_spread = pd.read_csv("test_spread.csv", header = 0, index_col = 0)
test_z_score = pd.read_csv("test_z_score.csv", header = 0, index_col = 0)
z_score_entry = pd.read_csv("z_score_optim.csv", header = 0, index_col = 0)

signal_df1 = []
entryZscore, exitZscore =  [0.8, -0.05]
for i in range(len(cointegration_pairs)):
    z_score = test_z_score.iloc[:, i]
    spread = test_spread.iloc[:, i]

    df = pd.DataFrame({"Z_score": z_score, "Spread": spread})
    df['long entry'] = (df["Z_score"] < (-entryZscore)) &
        (df["Z_score"].shift(1) > (-entryZscore))
    df['long exit'] = (df["Z_score"] > (-exitZscore)) &
        (df["Z_score"].shift(1) < (-exitZscore))
    df['short entry'] = (df["Z_score"] > (entryZscore)) &
        (df["Z_score"].shift(1) < (entryZscore))
    df['short exit'] = (df["Z_score"] < (exitZscore)) &
        (df["Z_score"].shift(1) > (exitZscore))
    signal_df1.append(df.dropna())


signal_df2 = []
for i in range(len(cointegration_pairs)):
    z_score = test_z_score.iloc[:, i]
    spread = test_spread.iloc[:, i]
    longEntry, longExit, shortEntry, shortExit = z_score_entry.iloc[i, 2:]

    df = pd.DataFrame({"Z_score": z_score, "Spread": spread})
    df['long entry'] = (df["Z_score"] < longEntry) & (df["Z_score"].shift(1) >
        longEntry)
    df['long exit'] = (df["Z_score"] > longExit) & (df["Z_score"].shift(1) <
        longExit)
    df['short entry'] = (df["Z_score"] > shortEntry) & (df["Z_score"].shift(1)
        < shortEntry)
    df['short exit'] = (df["Z_score"] < shortExit) & (df["Z_score"].shift(1) >
        shortExit)
    signal_df2.append(df.dropna())

datetime_list = []
for index in signal_df1[0].index:
    datetime_list.append(dt.datetime.strptime(index, "%Y-%m-%d").date())

weight_p = pd.read_csv("weight_p_df.csv", header = 0, index_col = 0)
weight_t = pd.read_csv("weight_t_df.csv", header = 0, index_col = 0)
```

[5]Please refer to the code: `Backtest_BackTrader.py` for full version.

```
class Strat_Equal_Weight_Fixed_Entry(bt.Strategy):

    def __init__(self):
        sizer = []
        for _ in range(len(cointegration_pairs)):
            sizer.append([1, 1])
        self.trading_size = sizer

    ... log function and notify_trade function omitted

    def next(self):

        if self.datas[0].datetime.date(0) in datetime_list:

           date_row = datetime_list.index(self.datas[0].datetime.date(0))

           for i in range(len(cointegration_pairs)):

                idx_0 = data_idx[cointegration_pairs[i][0]]
                idx_1 = data_idx[cointegration_pairs[i][1]]

                ratio = self.datas[idx_0].close[0] / self.datas[idx_1].close[0]
                size = self.broker.get_value() / len(cointegration_pairs) /
                    self.datas[idx_0].close[0]

                if signal_df1[i].iloc[date_row, 2]:
                    self.sell(data = self.datas[idx_0], size = size)
                    self.buy(data = self.datas[idx_1], size = size * ratio)
                    self.trading_size[i] = [size, size * ratio]
                elif signal_df1[i].iloc[date_row, 3]:
                    self.buy(data = self.datas[idx_0], size =
                        self.trading_size[i][0])
                    self.sell(data = self.datas[idx_1], size =
                        self.trading_size[i][1])
                elif signal_df1[i].iloc[date_row, 4]:
                    self.buy(data = self.datas[idx_0], size = size)
                    self.sell(data = self.datas[idx_1], size = size * ratio)
                    self.trading_size[i] = [size, size * ratio]
                elif signal_df1[i].iloc[date_row, 5]:
                    self.sell(data = self.datas[idx_0], size =
                        self.trading_size[i][0])
                    self.buy(data = self.datas[idx_1], size =
                        self.trading_size[i][1])
```

```
class Strat_Equal_Weight_Optimzed_Entry(bt.Strategy):

    def __init__(self):
        sizer = []
        for _ in range(len(cointegration_pairs)):
            sizer.append([1, 1])
        self.trading_size = sizer

    ... log function and notify_trade function omitted

    def next(self):

        if self.datas[0].datetime.date(0) in datetime_list:

           date_row = datetime_list.index(self.datas[0].datetime.date(0))

           for i in range(len(cointegration_pairs)):

                idx_0 = data_idx[cointegration_pairs[i][0]]
                idx_1 = data_idx[cointegration_pairs[i][1]]

                ratio = self.datas[idx_0].close[0] / self.datas[idx_1].close[0]
                size = self.broker.get_value() / len(cointegration_pairs) /
                    self.datas[idx_0].close[0]

                if signal_df2[i].iloc[date_row, 2]:
                    self.sell(data = self.datas[idx_0], size = size)
                    self.buy(data = self.datas[idx_1], size = size * ratio)
                    self.trading_size[i] = [size, size * ratio]
                elif signal_df2[i].iloc[date_row, 3]:
                    self.buy(data = self.datas[idx_0], size =
                        self.trading_size[i][0])
                    self.sell(data = self.datas[idx_1], size =
                        self.trading_size[i][1])
                elif signal_df2[i].iloc[date_row, 4]:
                    self.buy(data = self.datas[idx_0], size = size)
                    self.sell(data = self.datas[idx_1], size = size * ratio)
                    self.trading_size[i] = [size, size * ratio]
                elif signal_df2[i].iloc[date_row, 5]:
                    self.sell(data = self.datas[idx_0], size =
                        self.trading_size[i][0])
                    self.buy(data = self.datas[idx_1], size =
                        self.trading_size[i][1])
)
```

```
class Strat_Optimzed_Weight_Optimzed_Entry(bt.Strategy):

    def __init__(self):
        sizer = []
        for _ in range(len(cointegration_pairs)):
            sizer.append([1, 1])
        self.trading_size = sizer

    ... log function and notify_trade function omitted

    def next(self):

        if self.datas[0].datetime.date(0) in datetime_list:

           date_row = datetime_list.index(self.datas[0].datetime.date(0))

           for i in range(len(cointegration_pairs)):

                idx_0 = data_idx[cointegration_pairs[i][0]]
                idx_1 = data_idx[cointegration_pairs[i][1]]

                if np.isnan(weight_p.iloc[date_row, i]):
                    weight = 1 / len(cointegration_pairs)
                else:
                    weight = weight_p.iloc[date_row, i]

                ratio = self.datas[idx_0].close[0] / self.datas[idx_1].close[0]
                size = self.broker.get_value() * weight /
                    self.datas[idx_0].close[0]

                if signal_df2[i].iloc[date_row, 2]:
                    self.sell(data = self.datas[idx_0], size = size)
                    self.buy(data = self.datas[idx_1], size = size * ratio)
                    self.trading_size[i] = [size, size * ratio]
                elif signal_df2[i].iloc[date_row, 3]:
                    self.buy(data = self.datas[idx_0], size =
                        self.trading_size[i][0])
                    self.sell(data = self.datas[idx_1], size =
                        self.trading_size[i][1])
                elif signal_df2[i].iloc[date_row, 4]:
                    self.buy(data = self.datas[idx_0], size = size)
                    self.sell(data = self.datas[idx_1], size = size * ratio)
                    self.trading_size[i] = [size, size * ratio]
                elif signal_df2[i].iloc[date_row, 5]:
                    self.sell(data = self.datas[idx_0], size =
                        self.trading_size[i][0])
                    self.buy(data = self.datas[idx_1], size =
                        self.trading_size[i][1])
```

To obtain a more realistic backtest result, we assume 0.1% transaction cost and no slippage occur. For every buy and sell trade, we will submit a market order to the broker.

We repeat the following code to get the backtest results of different strategies:

```
portfolio_starting_value = pow(10, 9)

cerebro = bt.Cerebro()
cerebro.broker.setcash(portfolio_starting_value)
cerebro.broker.setcommission(commission = 0.001)  # 0.1% transaction cost

data_idx = {}
for pairs in cointegration_pairs:
    for ticker in pairs:
        if ticker not in list(data_idx.keys()):
            # price[ticker].iloc[training_split:].to_csv(f"data/{ticker}.csv")
            data = HLOCV_csv(dataname =  f"data/{ticker}.csv")
            data.plotinfo.plot = False
            cerebro.adddata(data, name = ticker)
            data_idx[ticker] = len(data_idx)

print("Equal Weight & Fixed Entry")
cerebro.addstrategy(Strat_Equal_Weight_Fixed_Entry)

cerebro.addsizer(bt.sizers.FixedSize, stake = 100000)
cerebro.addobserver(bt.observers.DrawDown)
cerebro.addanalyzer(btanalyzers.AnnualReturn, _name = "annual_return")
cerebro.addanalyzer(btanalyzers.SharpeRatio, _name = "mysharpe")

print("Portfolio starting value:", cerebro.broker.getvalue())
thestrats = cerebro.run()
print("Portfolio ending value:", cerebro.broker.getvalue())

thestrat = thestrats[0]
print('Annualized Return:', thestrat.analyzers.annual_return.get_analysis())
print('Sharpe Ratio:', thestrat.analyzers.mysharpe.get_analysis())
print()

cerebro.plot()
```

5.2 Graphs

After we run the backtest using BackTrader, we obtain the following plots showing portfolio value, cash value, profit and loss of each trade and the maximum drawdown of each tested strategies.

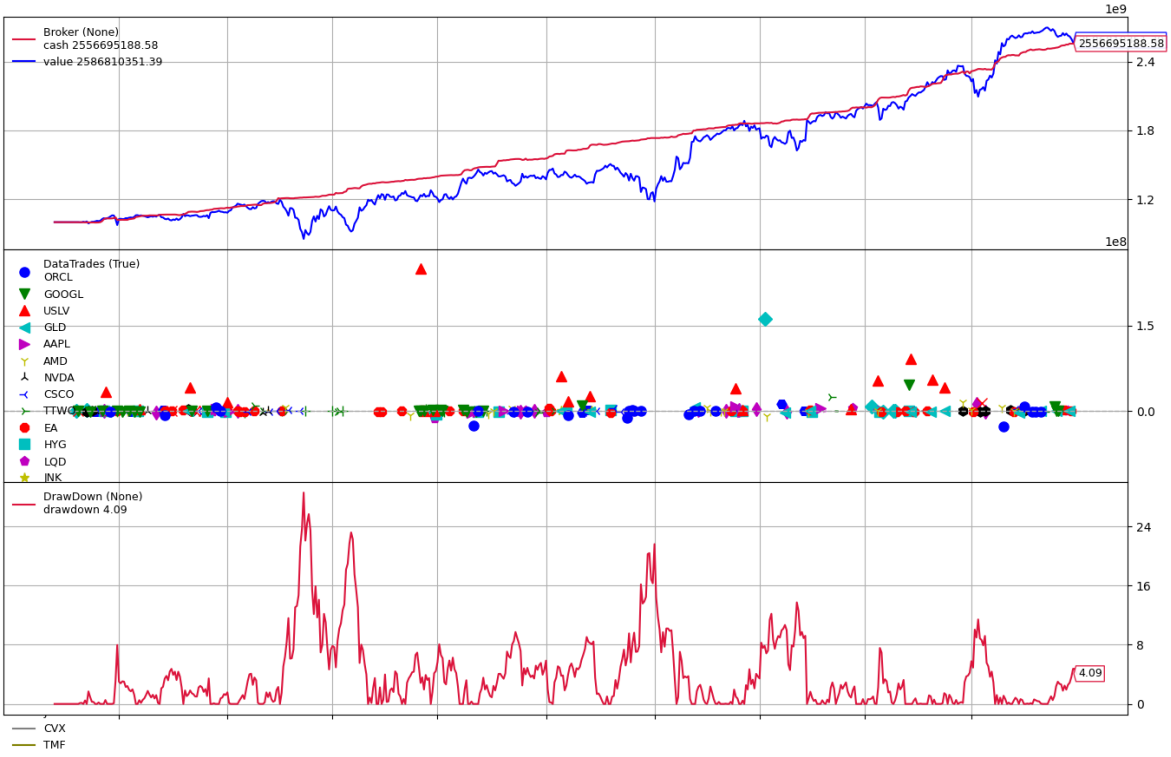


Figure 5: EW-FE

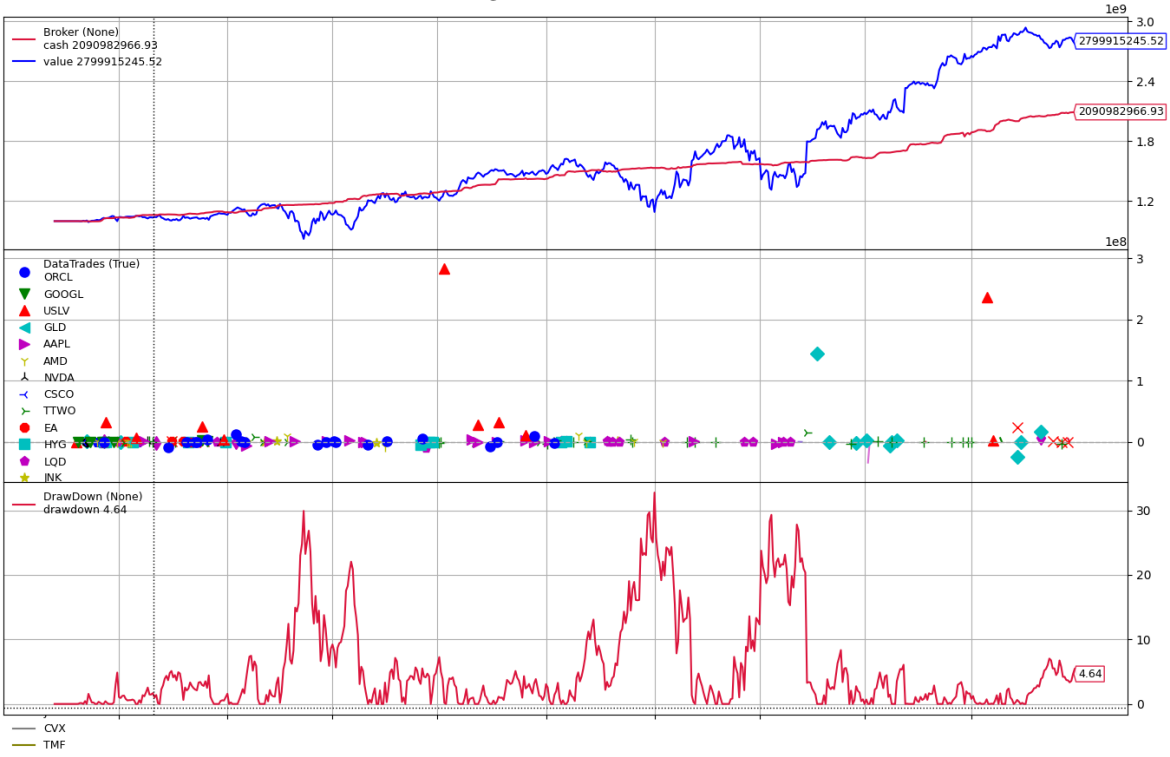


Figure 6: EW-OE

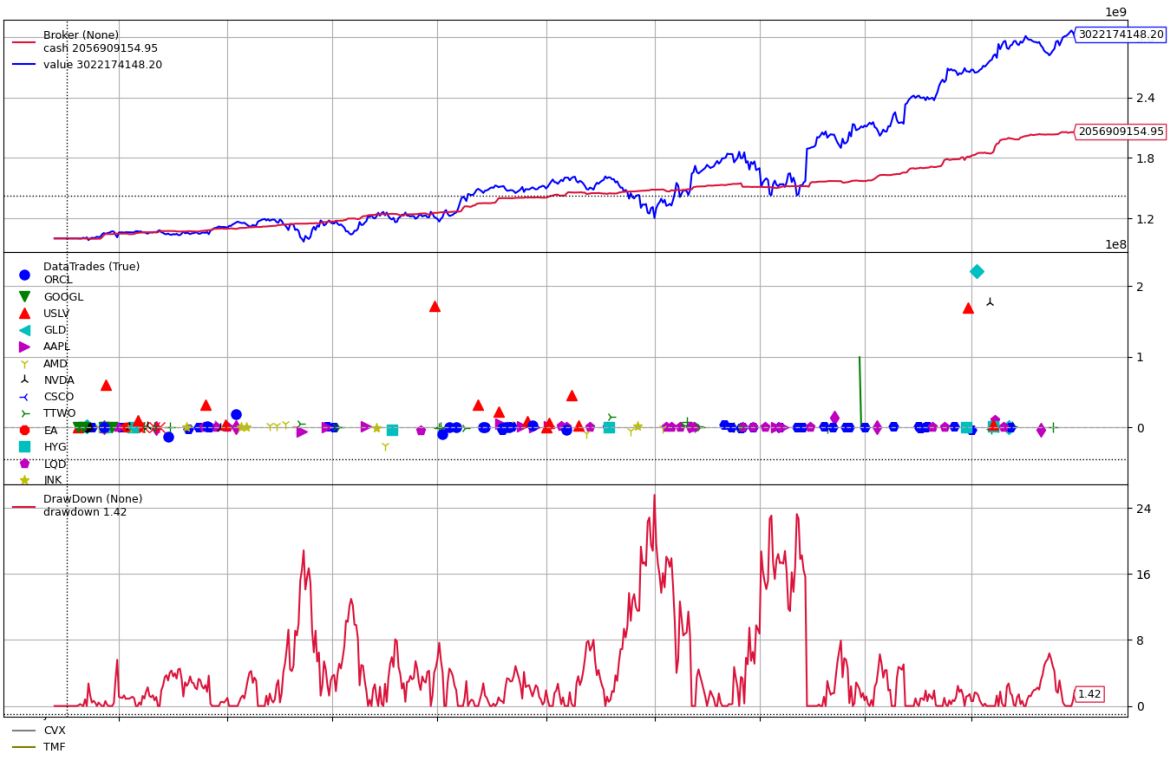


Figure 7: OW-OE

5.3 Return Metrics

	EW-FE	EW-OE	OW-OE
23/8/2021 - 31/12/2021	**8.26%**	**16.49%**	**105.11%**
1/1/2022 - 31/12/2022	**6.34%**	**14.48%**	**129.98%**
1/1/2023 - 31/12/2022	**10.78%**	**18.44%**	**130.34%**

Table 2: Return by year

	EW-FE	EW-OE	OW-OE
Sharpe Ratio	**0.96**	**0.87**	**0.95**
Maximum Drawdown	**28.58**	**32.75**	**25.59**
SQN	**3.68**	**2.72**	**3.28**
Number of Trades	**540**	**3.86**	**3.93**

Table 3: Strategy Metrics

6 Code Flow

Order	Program file name	Purpose of code
1	`Pairformation_HedgeRatio.py`	1. Get price data of the asset pool from Yfinance 2. Run co-integration test and find good co-integration pairs 3. Implement Kalmer filter on the price series to estimate β
2	`DCC_GARCH_model.R`	1. Optimize the entry and exit z-score threshold using DCC GARCH model
3	`z_score_optim.py`	1. A preliminary backtest to evaluate the effectiveness of the optimized z-score
4	`Weight_Calculation.py`	1. Calculate the optimal weight of each co-integration pairs
5	`Backtest_BackTrader.py`	1. Backtest trading strategies 2. Output graphs and return metrics

Table 4: Program Execution Order

7 Appendix

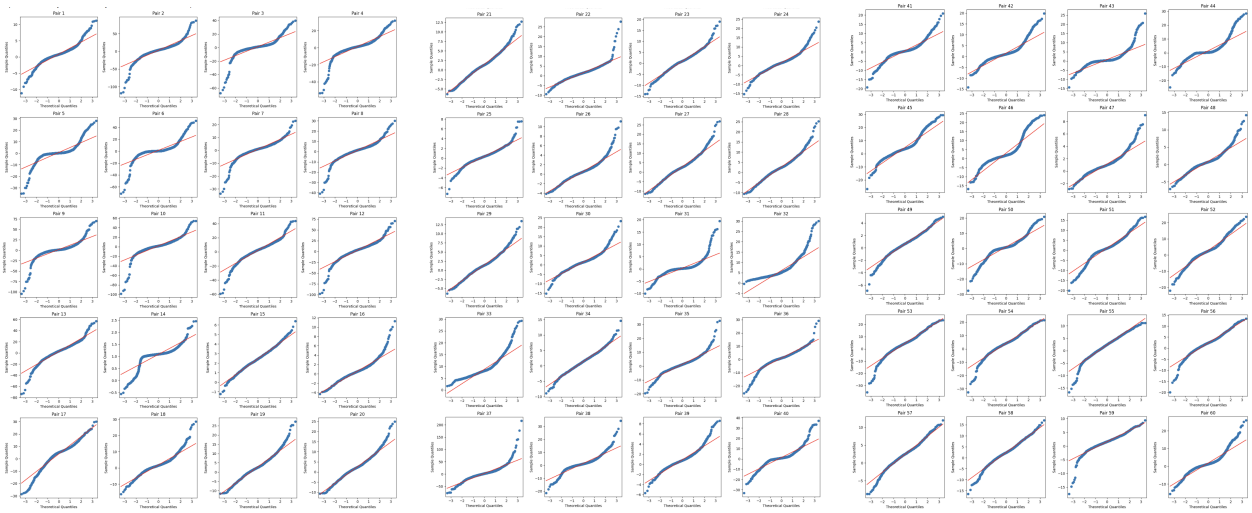
7.0.1 Head and Tail of optimized parameters for Max Profit and Max Sharpe

	MaxProfit	long_entryZscore	long_exitZscore	short_entryZscore	short_exitZscore
ORCL GOOGL	2.964471	-0.664773	0.050268	0.735625	-0.046282
USLV GLD	5.932841	-0.72478	0.057938	0.420411	-0.054521
USLV AAPL	7.173641	-0.409841	0.059464	0.737689	-0.053158
USLV GOOGL	7.926474	-0.812031	0.075921	0.030073	-0.051942
USLV AMD	5.738268	-0.327198	0.056442	0.819756	-0.064161
...	...	...	...	...	...
CMCSA VDC	3.108313	-0.616941	0.055844	0.598279	-0.053717
CMCSA KXI	3.509426	0.064196	0.055354	0.345365	-0.08867
CMCSA VHT	2.66107	-0.647945	0.053769	0.705584	-0.050185
CMCSA VNQ	3.243367	-0.559886	0.052759	0.737989	-0.053155
CMCSA IYR	3.033212	-0.702729	0.04865	0.740477	-0.049634

	MaxSharpe	long_entryZscore	long_exitZscore	short_entryZscore	short_exitZscore
ORCL GOOGL	2.142829	-0.665175	0.050439	0.745646	-0.048219
USLV GLD	1.959753	-0.82835	0.05173	0.483646	-0.052275
USLV AAPL	2.125589	-0.551336	0.053983	0.728412	-0.053249
USLV GOOGL	2.032549	-0.831308	0.051105	0.415312	-0.056196
USLV AMD	1.185272	-0.664329	0.046875	0.763904	-0.053986
...	...	...	...	...	...
CMCSA VDC	2.817447	-0.589789	0.058925	0.597373	-0.049333
CMCSA KXI	2.872766	-0.564648	0.053713	0.758276	-0.052278
CMCSA VHT	2.215425	-0.653663	0.05334	0.691434	-0.050305
CMCSA VNQ	2.624765	-0.690233	0.048605	0.748836	-0.048964
CMCSA IYR	2.531933	-0.703967	0.048533	0.75304	-0.048853

Figure 8: Left - Optimized Parameters for Max Profit, Right- Optimized Parameters for Max Sharpe

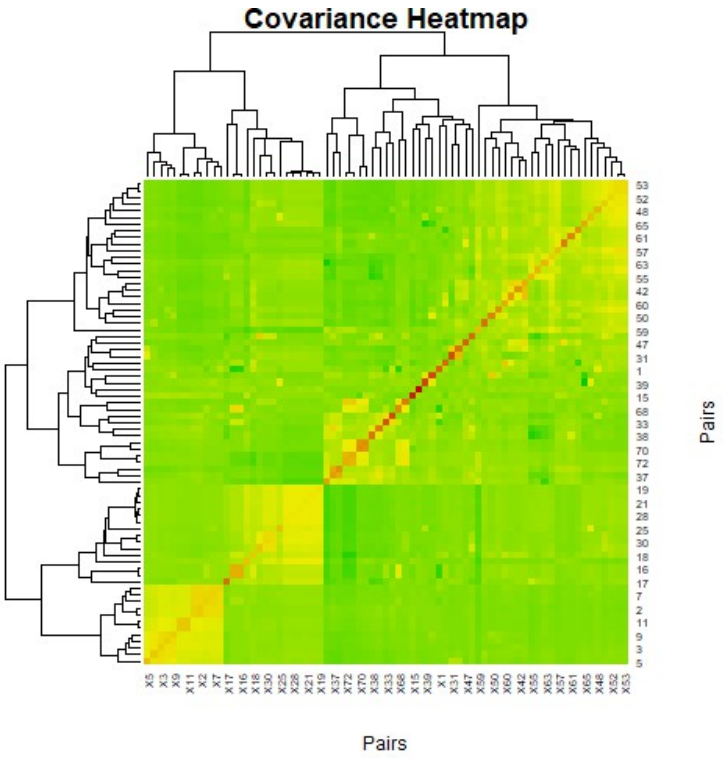
7.0.2 QQ plots on training spreads



7.0.3 Heatmap for Training spreads

7.0.4 CSV Files

- Click here to download test z score
- Click here to download train z score
- Click here to download optimized parameters with Maximum Profit
- Click here to download optimized parameters with Maximum Sharpe


Covariance Heatmap
Pairs
Pairs