# Cryptocurrency Portfolio Optimization

Simone Bonelli, Alessandro Ricchiuti, Alberto Yulian Hu, Babak Feyzullayev

July 6, 2024

## 1 Introduction

This project aims to implement portfolio optimization on a portfolio of cryptocurrencies. We start with a pool of cryptocurrencies, select a subset by Sharpe ratio, and perform mean-variance optimization on them. We also generate simulated data with Monte Carlo simulation and perform an additional analysis as a robustness check.

## 2 Methodology

To achieve our objective of optimizing a cryptocurrency portfolio, we employed the following approach:

1. Download historical price data of 100 cryptocurrencies.
2. Compute the correlation matrix and visualize it using a heatmap.
3. Select a subset based on a specified criterion.
4. Use the Gaussian KDE to find the empirical distribution that fits the data.
5. Generate random samples from the empirical distribution using Monte Carlo simulation.
6. Define and solve the optimization problem to find the optimal portfolio weights.

## 3 Data Collection and Processing

We started by collecting historical data for 100 cryptocurrencies from Yahoo Finance. The selection of data post-COVID-19 avoids the extreme volatility observed during the early months of the pandemic. The data was downloaded using the yfinance library:

```
data = yf.download(cryptos, start='2020-06-01', end='2023-01-01', interval='1d')['Adj Close']
```

The daily returns were calculated as the percentage change in the closing prices, then we removed any resulting NaN values:

```
returns = data.pct_change().dropna()
```

### 3.1 Correlation Matrix

The correlation matrix was calculated to understand the relationships between the returns of different cryptocurrencies. The heatmap of the correlation matrix is shown in Figure 1. There are noticeable clusters where groups of cryptocurrencies have higher correlations with each other. This indicates that certain cryptocurrencies tend to move together in the market, possibly due to similar market influences or being in the same category or sector. We computed the correlation matrix of the returns DataFrame and visualized it as a heatmap using the Seaborn library:

```
corr_df = returns.corr()
sns.heatmap(corr_df)
```

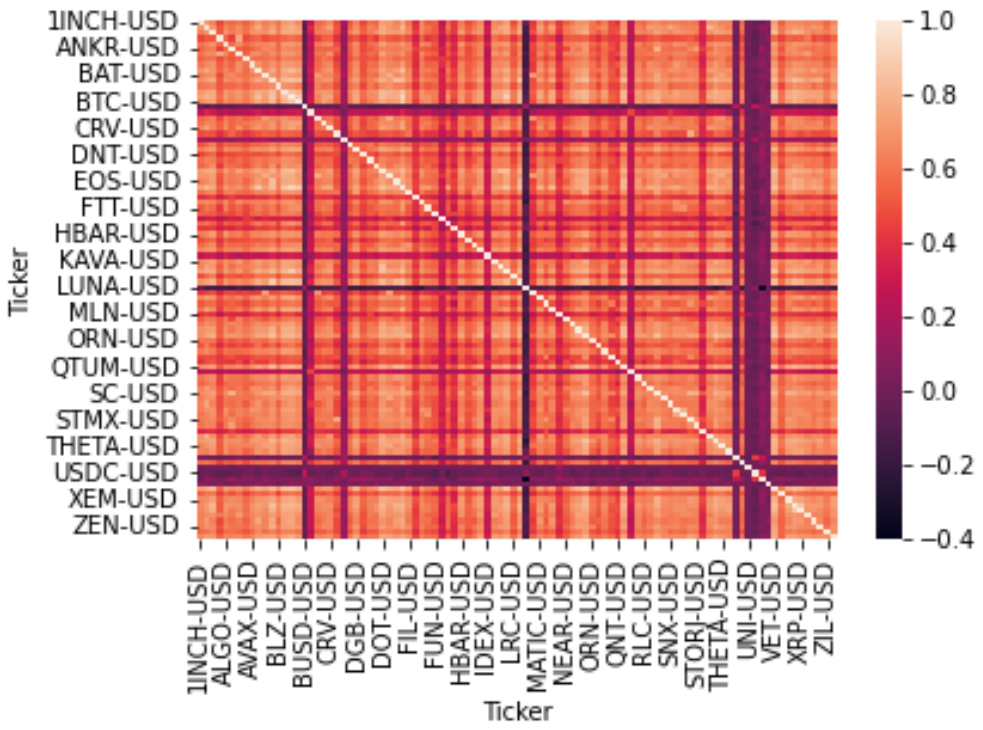


Figure 1: Correlation Matrix Heatmap

## 4 Sharpe Ratio Calculation

The Sharpe Ratio for each asset was calculated using the following function:

```
def calculate_sharpe_ratios(df, risk_free_rate_df):
    sharpe_ratios = {}

    for column in df.columns:
        asset_returns = df[column]
        risk_free_rate = risk_free_rate_df.reindex(asset_returns.index).ffill().bfill()
        excess_returns = asset_returns - risk_free_rate
        mean_excess_return = excess_returns.mean()
        std_dev = excess_returns.std()
        sharpe_ratio = mean_excess_return / std_dev
        sharpe_ratios[column] = sharpe_ratio

    sharpe_ratios_df = pd.DataFrame(list(sharpe_ratios.items()), columns=['Asset', 'Sharpe Ratio'])

    return sharpe_ratios_df
```

The top 10 assets with the highest Sharpe Ratios were selected. We should note that this doesn't hold any value from a practitioner's perspective, as unless there is some evidence that such assets exhibit momentum-like behaviors, out of sample, they are likely to not outperform or even perform worse than the others if some mean-reverting relation holds.

# 5 Empirical Distribution

As an example, we plot the empirical distribution of USDC, which is shown in Figure 2, and overlap a normal distribution on it:

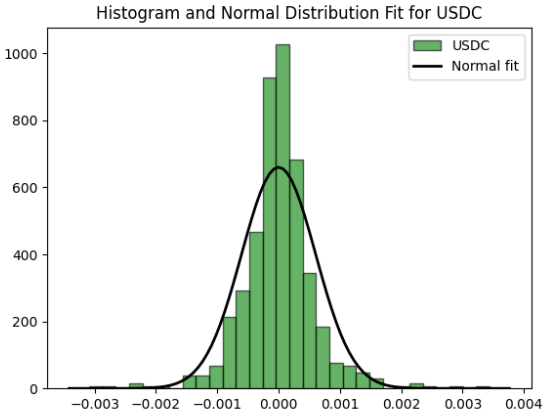


Figure 2: Empirical distribution of USDC returns, overlapped with a normal distribution

We notice that the normal distribution does not fit the data well, so we resort to Kernel density estimation to find a suitable distribution.

## 5.1 Kernel Density Estimation (KDE)

We apply KDE to the returns data to obtain a smoother and more accurate estimate of the distribution. We use the Gaussian KDE method from scipy.stats to fit the empirical data.

```
kde = gaussian_kde(USDC)
```

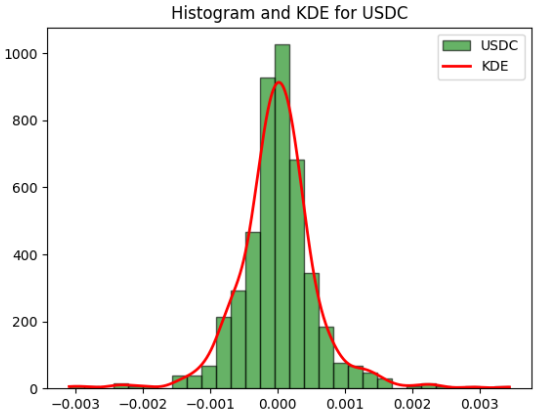


Figure 3: Empirical distribution of USDC returns, overlapped with a kernel density estimation

# 6 Monte Carlo Simulation

Now that we have a suitable density to generate from, we use Monte Carlo simulation to generate data for our target distribution. We use the acceptance-rejection method with the uniform distribution as an

envelope. We make sure that the support of the target distribution and the envelope are the same, so we generate according to a uniform distribution with support defined from the minimum value to the maximum of the empirical distribution. The number of generated observations per asset is equal to the sample size of the empirical distributions.

```
def rejection_sampling(kde, min_val, max_val, num_samples=600):
    samples = []
    max_density = np.max(kde(x))
    while len(samples) < num_samples:
        u = np.random.uniform(min_val, max_val)
        v = np.random.uniform(0, max_density)
        if v < kde(u):
            samples.append(u)
    return samples
```

Samples are proposed from a uniform distribution over the range from the minimum to the maximum. The values are then accepted based on the value of the KDE evaluated at the proposed sample compared to a uniformly drawn threshold. With this method, we can generate simulated data according to the Kernel density we estimated before.

# 7 Portfolio Optimization

We introduce some functions that will allow us to perform portfolio optimization.

## 7.1 Function Definitions

### 7.1.1 Portfolio Return

This function calculates the expected annual return of the portfolio. It computes the weighted sum of the expected returns using the dot product of the weights and returns, and scales this result to an annual basis.

### 7.1.2 Portfolio Variance

This function calculates the annual variance (risk) of the portfolio. It uses the covariance matrix of the asset returns and computes the portfolio variance with the weights of the portfolio. The result is scaled to an annual basis.

### 7.1.3 Portfolio Volatility

This function calculates the annual volatility (standard deviation) of the portfolio. It calls the portfolio variance function to obtain the variance and then takes the square root of the variance to get the standard deviation.

### 7.1.4 Portfolio Return Constraint

This function defines a constraint to ensure the portfolio achieves a target return. It calculates the expected annual return of the portfolio and subtracts the target return from this value. This constraint is used in optimization to ensure the portfolio meets or exceeds the specified target return.

## 7.2 Real and Simulated Data Optimization

Portfolio optimization is performed on real and simulated data using the Mean-Variance Optimization approach. The goal is to find the optimal weights of the assets that minimize the portfolio variance while achieving a target return.

```
annual = 252  # Annualization factor for daily returns
x0 = pd.Series(1/selected_df.shape[1], index=selected_df.columns)  # Initial equal weights for all asse
mu = selected_df.mean()  # Mean returns of the assets
r = selected_df.copy()  # Copy of the returns DataFrame

expected_return = port_ret(x0, mu, annual)  # Calculate the expected annual return of the initial portfo
initial_variance = port_variance(x0, r, annual)  # Calculate the initial portfolio variance
initial_volatility = port_vola(x0, r, annual)  # Calculate the initial portfolio volatility

mu_0 = 0.09  # Target annual return
cons_MV = ({'type': 'eq', 'fun': lambda x: sum(x) - 1},
           {'type': 'eq', 'fun': port_ret_eq, 'args': (mu, annual, mu_0)})

res = spopt.minimize(port_variance, x0, method='SLSQP', args=(r, annual),
                     constraints=cons_MV, options={'disp': True})
```

**Explanation:**

- **Annualization Factor:** The variable `annual` is set to 252, representing the number of trading days in a year, used to annualize daily returns.
- **Initial Weights:** `x0` initializes the portfolio weights equally among all assets.
- **Mean Returns:** `mu` calculates the mean returns of the assets from the `selected_df` DataFrame.
- **Returns DataFrame:** `r` is a copy of the returns DataFrame.
- **Initial Metrics:** The expected annual return, variance, and volatility of the initial portfolio are calculated using the `port_ret`, `port_variance`, and `port_vola` functions, respectively.
- **Target Return:** `mu_0` is set to 0.09, representing the target annual return.
- **Constraints:** `cons_MV` defines the optimization constraints, ensuring that the sum of the weights equals 1 and the portfolio achieves the target return.
- **Optimization:** The `spopt.minimize` function is used to minimize the portfolio variance subject to the defined constraints using the Sequential Least Squares Programming (SLSQP) method.

## 7.3 Real Data Output

- **Optimized Portfolio Weights:**

```
[ 0.9614,  0.2579,  0.0868, -0.0638, -0.2877,
  0.0264,  0.0147,  0.0003,  0.0000,  0.0040 ]
```

- **Optimized Portfolio Return:** 0.08999999994842244
- **Optimized Portfolio Variance:** 0.003751311922088407
- **Optimized Portfolio Volatility:** 0.06124795443186986

## 7.4 Simulated Data Output

- **Optimized Portfolio Weights:**

```
[ 0.317130,  0.374646, 0.501116, -0.006499, -0.191034,
  0.002228, 0.000902,  0.000075,  0.000001,  0.001435 ]
```

- **Optimized Portfolio Return:** 0.08999999993932595
- **Optimized Portfolio Variance:** 0.00028615930957293204
- **Optimized Portfolio Volatility:** 0.01691624395582341

## 7.5 Comments

From these results, we gain an understanding of how sensitive mean-variance optimization is to input estimation. In a paper by Best, Michael and Grauer, Robert. (1991), the authors discuss that it takes surprisingly small changes in the covariances of assets to change the composition of the portfolio. In our analysis, as shown in Figures 4 and 5, even though the shape of the empirical distribution is identical between real and simulated data, the covariance structure between the assets is not exactly replicated in the simulated data; it has extreme sensitivity. This bears important consequences: working with simulated data can omit vital details. We notice that the portfolio variance in the simulated data is almost 20 times lower, something which changes the analysis completely.

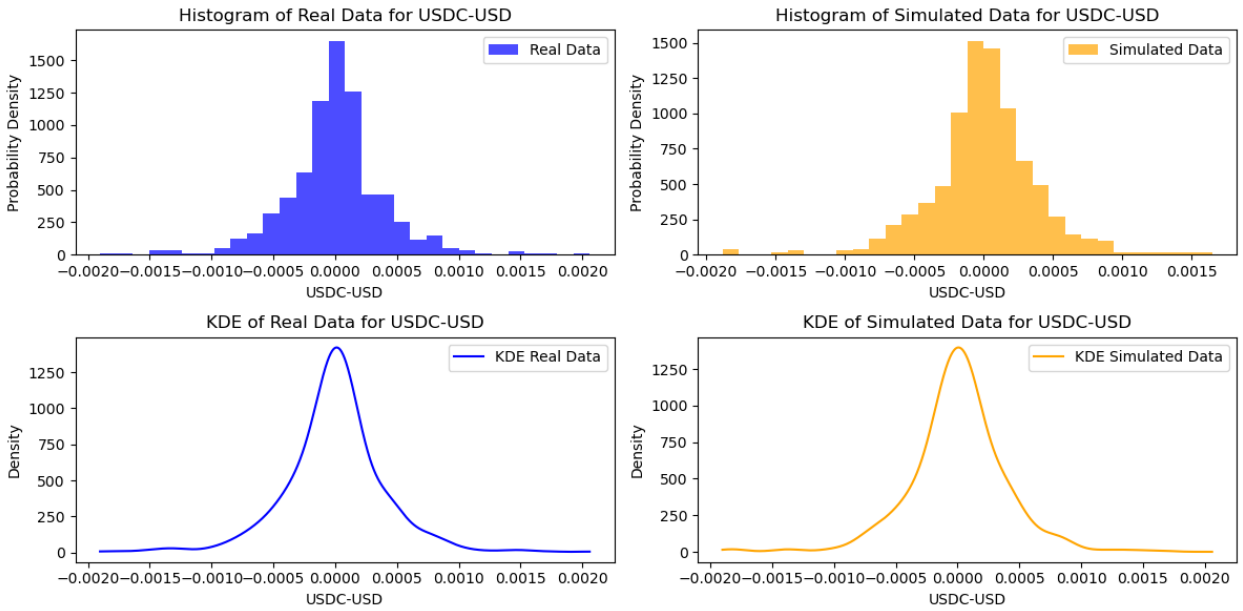


Figure 4: Empirical distribution and KDE of USDC-USD

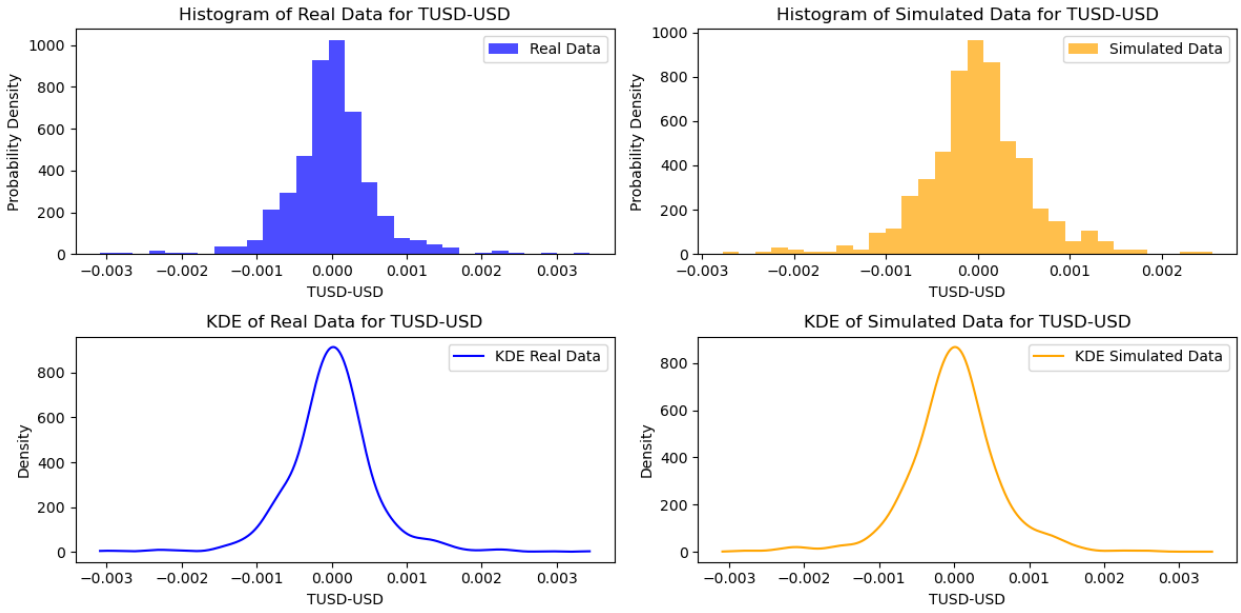


Figure 5: Empirical distribution and KDE of TUSD-USD

# 8 Conclusion

This project has outlined an approach to optimizing a cryptocurrency portfolio, using historical price data, analyzing their correlations, fitting empirical distributions with Gaussian KDE, and using Monte Carlo simulations to generate data samples.
Our study revealed how sensitive mean-variance optimization is to the inputs, especially the means and covariances of asset returns. By comparing real and simulated data, we saw significant differences in portfolio variances and weights.

# 9 Bibliography

Best, Michael & Grauer, Robert. (1991). On the Sensitivity of Mean-Variance-Efficient Portfolios to Changes in Asset Means: Some Analytical and Computational Results. Review of Financial Studies. 4. 315-42. 10.1093/rfs/4.2.315.