L1: Portfolio Returns

if the correlations between two assets increases/decreases, the portfolio variance will increase/decrease. Subsequently, if the correlations of many assets increase at the same time, the portfolio variance will increase faster, and that could be an indication of systemic risk.

L2: Efficient Frontier, Sharpe Ratio and Downside Risk

1.2 Adjusted-Close Price

A simple way of calculating the adjusted-close (assuming no splits) is the following:

$$p_t^{\text{adj}} = p_t^{\text{close}} - \sum_{i>t} D_i$$

which essentially means that at any given time t, we need to subtract all dividends that are paid from time t and on. Calculating the adjusted-close is a backwards-looking procedure. For example:

Year	Close Price ($)	Dividends($)	Adjusted Close ($)
2019:	105	2	105 - 2.2 - 2.3 - 2.4 = 98.10
2020:	112	2.2	112 - 2.3 - 2.4 = 107.30
2021:	118	2.4	118 - 2.3 = 115.07
2022:	116	2.3	116

So now, let's calculate the dividend-adjusted returns and the adjusted-close returns over the period of 2019 to 2021.

$$\text{Dividend Adjusted Returns} = \frac{(118 - 105) + 2.4 + 2.2}{105} = 0.1676$$

and

$$\text{Adjusted Close Returns} = \frac{115.07 - 98.10}{98.10} = 0.1723$$

The above two numbers are different. Do you know why?

The reason lies in the reinvestment assumption when calculating the adjusted-close price. One can see that this transformation leads to significant changes in the price array: for 2019, the observed price was 105 whereas the adjusted-close was 98.10. Similarly, for 2021, the observed price was 118 while the adjusted-close was 115.07. The difference between the two adjusted-close prices is attributed to price appreciation (the same as with the close prices) and the dividends paid during that period. In essence, the share's close price along with the dividends **is replicated with a price that incorporates the dividends as if the latter was a part of price appreciation**.

But if the price of an asset one holds appreciates, then the appreciation is still part of the asset and not cash as is the case when dividends are paid. In this observation, we find an important distinction between the dividend-adjusted returns and adjusted-close returns: when using adjusted-close returns, we assume that we reinvest the dividends in more shares.

For a more detailed and precise way of calculating the adjusted close, please read this: https://help.yahoo.com/kb/SLN28256.html?guccounter=1. Make sure you can explain the benefits of using multipliers instead of subtracting the dividends in adjusting the price.

---

2. Excess Returns

2.1 Excess Returns Definition

As you already remember from the Financial Markets course, excess return is the difference between the returns of an asset (or portfolio) and a benchmark. An analyst can use various benchmarks depending on the analysis, but the most common ones are a market proxy (S&P500) or a risk-free rate (T-Bills). The formula for excess returns is:

$$\text{Excess Returns} = r_{\text{portfolio}} - r_{\text{benchmark}}$$

where:

- $r_{\text{portfolio}}$ is the **total return** of the investment
- $r_{\text{benchmark}}$ is the returns of the benchmark

In calculating excess returns, it is crucial to account for all types of cash flows associated with the investment, which is why total returns are used. In practice, however, adjusted-close returns can be used, provided that any additional cash flows are manually included if they are not already factored into the adjusted-close price by the data provider.

But now, we need to talk about the benchmark. As we have mentioned, the benchmark is case specific and can be represented by any asset an analyst decides to test the returns of their investment against. For example:

- Selecting a benchmark in terms of objectives, strategy, and sector: S&P500 as a broad market benchmark, NASDAQ for technology and growth portfolios, or Russell 2000 for small-cap equity portfolios.
- Geographic exposure: the benchmark should represent the region where the investment takes place. In the UK, one could use FTSE 100 or FTSE 250; in the EU, there is EURO STOXX 50, DAX, or CAC 40; and in India, one has NIFTY 50, S&P BSE 500, and more.
- Using the risk-free rate as the benchmark, one should take into account the:
  - Investment horizon: for short-term investments, government securities like 3-month T-bill are appropriate. For longer-term investments, 10-year treasuries could be more

2.2 Excess Returns over Expected Return

In the context of finance, the expected return of an asset represents the return an investor expects/requires given the risk profile of the specific asset. That means that in calculating the expected return, one can take into account the systematic risk, inflation expectations, interest rate risk, and more. If the actual returns of the asset are greater than expected returns, then we can say that *the asset has outperformed expectations*.

$$\text{Excess Returns} = r_{\text{actual}} - \mathbb{E}(r_{\text{asset}})$$

But calculating the expected return is not straightforward and depends on several strong assumptions. We will mention several methods used to estimate the expected returns of an asset. These methods are distinguished with respect to the types of risk that they make use of:

CAPM

A widely used model is the capital asset pricing model (CAPM), which models returns with respect to systematic risk. The formula for CAPM is:

$$\mathbb{E}(r_{\text{asset}}) = r_f + \beta(r_m - r_f)$$

where

- r_f is the risk-free rate
- r_m is a market proxy
- β is the sensitivity of the asset to the market returns.

In the CAPM world, the excess return is often called *alpha* (α):

$$\alpha = r_{\text{actual}} - \left(r_f + \beta(r_m - r_f)\right)$$

APT

The arbitrage pricing theory (APT) goes beyond the CAPM model in incorporating multiple factors of risk instead of just the systematic one (market). These factors could be any macroeconomic variable such as inflation, interest rates, economic growth, etc. The formulation is:

$$\mathbb{E}(r_{\text{asset}}) = r_f + \beta_1 f_1 + \beta_2 f_2 + \cdots + \beta_n f_n$$

where

- f_i are the factor risk premiums
- β_i are the sensitivities of the asset to these factors.

Fama-French Three-Factor Model

This model extends CAPM by adding two additional factors:

1. The *SMB* (small minus big) factor, which represents the "size premium." This is essentially the difference between small-cap and large-cap stocks.
2. The *HML* (high minus low) factor, which measures the difference between the returns of the value and growth stocks.

As with the CAPM and APT models, Fama-French also assumes linearity between expected returns and factors:

$$\mathbb{E}(r_{\text{asset}}) = r_f + \beta_{\text{market}}(r_m - r_f) + \beta_{size} \cdot \text{SMB} + \beta_{\text{value}} \cdot \text{HML}$$

The three models above were merely mentioned to explain the notion of expected returns and its importance in calculating excess returns. Essentially, excess returns measure performance over a chosen benchmark, which can be an external index or the asset's own expected return based on models like CAPM or APT.

3. Portfolio Sharpe Ratio and the Efficient Frontier

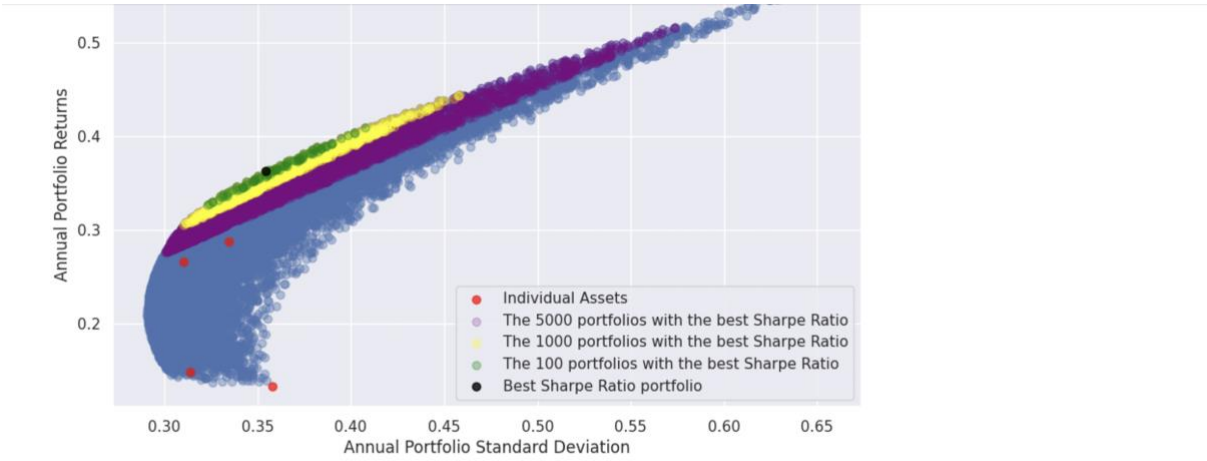
We have already mentioned what the Sharpe ratio is in the previous module. We will now illustrate the importance of the Sharpe ratio in the context of portfolio management. In order to do so, we will assume the assets of Lesson 1, and we will show where the highest Sharpe ratio portfolios are located with respect to the efficient frontier.

[5]:
```
assets = ['MSFT', 'AAPL', 'AMZN', 'TSLA', 'GOOGL'] # Assets for portfolio

asset_prices = yf.download(assets, start='2018-01-01', end='2023-01-01')['Adj Close'] # Downloading daily data
asset_prices.index = pd.to_datetime(asset_prices.index.date)

# Download risk-free rate data (3-month Treasury bill)
risk_free = yf.download('^IRX', start='2018-01-01', end='2023-01-01')['Adj Close']
risk_free.index = pd.to_datetime(risk_free.index.date)


r = asset_prices.pct_change().dropna() # Calculating daily percent returns
risk_free
```

```
[**********100%**********]  5 of 5 completed
[**********100%**********]  1 of 1 completed
```

[5]:

Ticker	^IRX
2018-01-02	1.378
2018-01-03	1.370
2018-01-04	1.370
2018-01-05	1.370
2018-01-08	1.380
...	...



What we essentially see in Figure 2 is the efficient frontier, as explained in the previous lesson, onto which we have additionally plotted the highest Sharpe Ratio portfolios.

More specifically, we illustrate where the 5000, 1000, and 100 portfolios with the highest Sharpe ratios are located and of course where the best Sharpe ratio portfolio is. As expected (?), those portfolios are on or very close to the efficient frontier, but to our surprise, they do not occupy the entire efficient frontier curve.

4. The Tangency Portfolio and the Tangent Line

Given the above observations, we can now give a definition for the best Sharpe ratio portfolio: the best Sharpe ratio portfolio, or the tangency portfolio, is the one that maximizes returns per unit of risk. **It provides the best risk-return tradeoff among all portfolios** (Elton et al.).

Our discussion and illustrations of portfolios' efficient frontiers give the impression that the efficient frontier contains optimal portfolios. In the next section, we will discuss the capital market line in order to illustrate how the tangency portfolio along with the risk-free rate can be used in creating a series of portfolios that are superior to the efficient frontier.

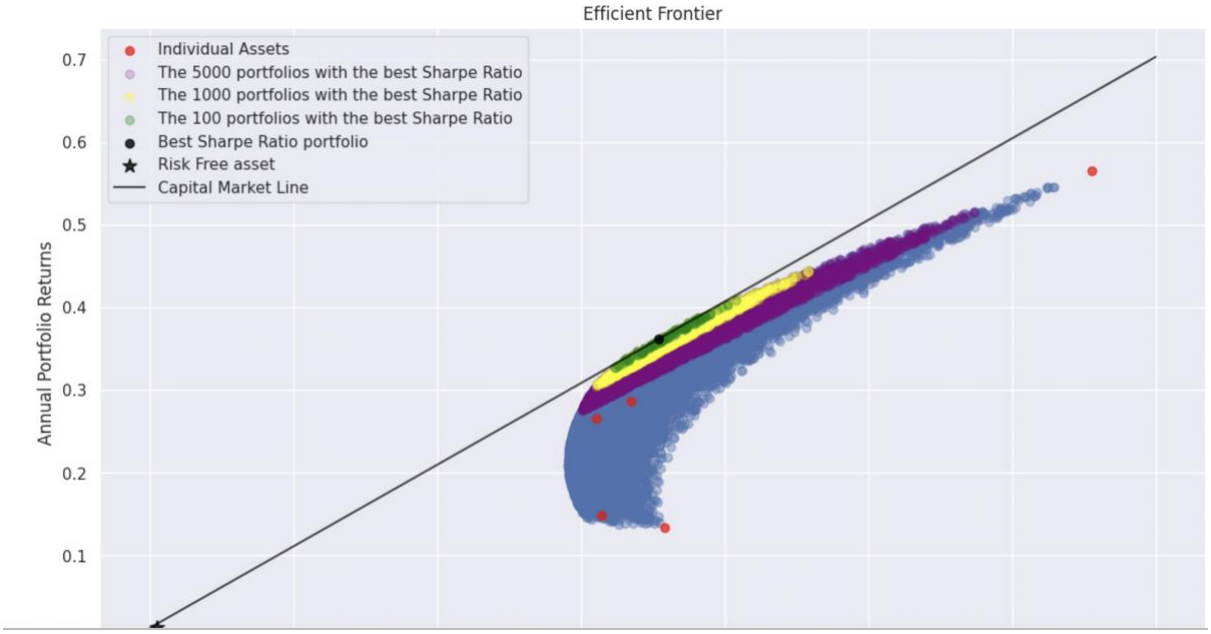
The tangent line, or the capital market line, is a key concept that arises from combining a risk-free asset with the tangency portfolio. Mathematically, the CML formulation is:

$$r_{\text{CML}} = r_f + \frac{r_T - r_f}{\sigma_T} \cdot \sigma_P$$

where

- r_{CML} is the expected return of any portfolio on the CML
- r_T is the return of the tangency portfolio (best Sharpe ratio portfolio)
- σ_T is the std of the tangency portfolio
- σ_p is the std of any portfolio on the CML

Let's visualize the CML before providing some intuition behind its use:



the best Sharpe ratio portfolio = the tangency portfolio

First of all, we need to observe that **the slope of the CML line is the Sharpe ratio** of the tangency portfolio and that the CML represents a crucial improvement over the efficient frontier. But how are the portfolios of the CML constructed? Let's rearrange the CML formulation as such:

$$r_{\text{CML}} = r_f \left(1 - \frac{\sigma_P}{\sigma_T}\right) + r_T \left(\frac{\sigma_P}{\sigma_T}\right)$$

Simply by combining the risk-free asset with the tangency portfolio and assigning the next weights:

- Tangency portfolio weight: $w_T = \frac{\sigma_P}{\sigma_T}$
- Risk-free weight: $w_f = 1 - \frac{\sigma_P}{\sigma_T}$

There will be more on this in the Portfolio Management course.

5. Value at Risk (VaR)

Knowing the variance and standard deviation is a big help when it comes to risk. Doing so can help us calculate one of the several ways we can measure risk by using **Value at Risk**.

Value at Risk is one of the easiest risk metrics to interpret. So far, the metrics we have introduced quantify risk as a percentage, in the case of standard deviation, or in units, as the Sharpe ratio does. Value at Risk answers the fundamental question many investors have on their mind: how much can I lose on an investment in the worst-case scenario? This is measured in dollars for the purposes of this class.

Value at Risk measures the potential loss in value of an asset/portfolio over a defined time period. Basically, you will always need to specify the time period and confidence interval that accompanies a Value at Risk. For example, if the VaR of a portfolio is $1,000,000 over a yearly time period with a 99% confidence interval, it would mean that the portfolio only has a 1% chance of losing more than $1,000,000 for any given year. VaR has become ubiquitous over the years; every investment bank and risk management firm employs some form of VaR to help keep a cap on the potential losses one can incur. The focus on VaR is very much about downside risk unlike something like standard deviation, which looks at both the upside and downside risk.

There are three basic methods for calculating VaR, each with their own advantages and disadvantages. These build on some of the lessons from earlier, like variance and covariance. Keep in mind that there are countless variations of each basic method, but we will stick with these main three for now:

- Historical Method
- Parametric Method
- Monte Carlo Simulation

Value at Risk measures the potential loss in value of an asset/portfolio over a defined time period. Basically, you will always need to specify the time period and confidence interval that accompanies a Value at Risk.

For example, if the VaR of a portfolio is $1,000,000 over a yearly time period with a 99% confidence interval, it would mean that the portfolio only has a 1% chance of losing more than $1,000,000 for any given year

6. VaR Using the Historical Method

This is probably the simplest and most intuitive method of calculating Value at Risk. In short, historical returns are sorted from lowest to highest on an asset or portfolio. Let's say you want to calculate the daily Value at Risk on an equity with a 95% confidence interval. Assuming we can look at the last thousand days of data for this stock, we would take the daily returns and sort them from lowest to highest. From here, we would take the return from the 5th percentile of the data. In this case, with 1,000 days of data, it would be the 50th (0.05*1000) worst daily return from these thousand days. Let's say the 50th worst day had a −4% return. From this, we can assume that the daily VaR for this stock with a 95% confidence interval is −4%. Building on that, if we were to invest $1,000 in said stock, we would expect the worst daily loss to be:

-0.04*1000 = -$40 with a 95% confidence interval

Considerations for this method:

- This method uses historical returns to measure VaR empirically, which means that there are no assumptions made about the distribution whereas many models assume the normal distribution.
- Each day for this method is given equal weight, which means if there is a trend in the volatility, you could be overstating or understating the VaR depending on whether the volatility trend is down or up, respectively. One refinement to combat this could be to place greater weight on more recent data.
- Past data does not necessarily indicate what will happen in the future. While the other methods also rely on historical data to a certain extent, this method is solely derived from past historical returns. There are many unforeseen events that can happen, which can change the course of a stock's trajectory and cause the stock to trade differently than it did in the past.

6.1 Implementing Value at Risk (VaR) - Historical Method

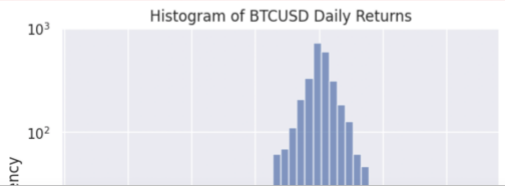
Calculating daily historical VaR can be done pretty simply in Python. The order of steps needed is as follows:

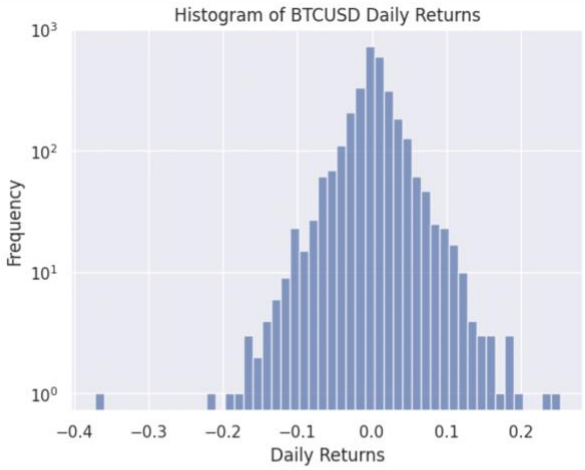
1. Calculate all daily returns.
2. Sort these returns from least to greatest.
3. Based on a given confidence level, return the corresponding percentile return. In other words, if we want to calculate daily VaR at a 95% confidence level and we are using 100 data points, the 5th smallest return of that sample would be considered our 95% VaR.

Let's visualize this first with a histogram using Bitcoin.

12]:
```
# Figure 1
btcusd = yf.download("BTC-USD", start="2014-09-18", end="2022-12-31")['Adj Close']
btcusd_returns = btcusd.pct_change().dropna()

plt.hist(x = btcusd_returns, bins=50, log = True, alpha = 0.7)
plt.xlabel('Daily Returns')
plt.ylabel('Frequency')
plt.title('Histogram of BTCUSD Daily Returns')
plt.show()
```

```
[*********************100%***********************]  1 of 1 completed
```





In order to have a better visualization of the extreme events, we used a logarithmic scale for the y axis. Let's calculate the historical VaR from this data.

The next function `getHistoricalVar()` takes two arguments:

1. a series of daily returns
2. a confidence level, 95 for example, would correspond to 95% confidence

In order to have a better visualization of the extreme events, we used a logarithmic scale for the y axis. Let's calculate the historical VaR from this data.

The next function `getHistoricalVar()` takes two arguments:

1. a series of daily returns
2. a confidence level, 95 for example, would correspond to 95% confidence

13]:
```
def getHistoricalVar(returns, confidenceLevel):
    var = np.quantile(returns, 1 - confidenceLevel)
    print(
        f"The Historical VaR with confidence {confidenceLevel} is {round(100 * var, 3)}% \n"
    )

getHistoricalVar(btcusd_returns, 0.90)
getHistoricalVar(btcusd_returns, 0.95)
getHistoricalVar(btcusd_returns, 0.99)
```

```
The Historical VaR with confidence 0.9 is -3.717%

The Historical VaR with confidence 0.95 is -6.0%

The Historical VaR with confidence 0.99 is -10.57%
```

After running the function, we can see that the greater we make our confidence level, the lower the Value at Risk will be. This is a useful tool when determining how much a financial asset can lose over a certain time period.

You may be wondering now, since we only use a 95% confidence level, what happens on those 5% of days where losses exceed -6.02%. We can use another handy metric, Conditional Value at Risk (CVaR), to deal with these situations.

6.2 Conditional Value at Risk (CVaR) using Historical Data

Conditional Value at Risk (CVaR), also known as Expected Shortfall (ES), is a widely used risk measure in financial risk management. While Value at Risk (VaR) estimates the loss over a specific time horizon given a confidence level (e.g., 99%), CVaR goes beyond VaR by providing the expected loss if the VaR is exceeded. In other words, CVaR calculates the average loss in the tail of the distribution beyond the VaR threshold.

Mathematically, the CVaR at confidence level α is:

$$\text{CVaR}_\alpha = \mathbb{E}[L | L > \text{VaR}_\alpha]$$

where L represents the losses.

Historical CVaR is a non-parametric approach, which means that it does not rely on distribution assumptions for the returns. Instead, we can directly calculate the expectation of the losses after we have obtained the VaR. In practice, we can calculate the mean of the returns that are located to the left of the VaR.

```
]: def getHistoricalCVar(returns, confidenceLevel):
       var = np.quantile(returns, 1 - confidenceLevel)
       cvar = returns[returns <= var].mean()
       print(
           f"With {confidenceLevel} percent confidence VaR, our Expected Shortfall is {round(100 * cvar, 3)} using historical VaR \n"
       )

   getHistoricalCVar(btcusd_returns, 0.90)
   getHistoricalCVar(btcusd_returns, 0.95)
   getHistoricalCVar(btcusd_returns, 0.99)
```

CVaR calculates the average loss in the tail of the distribution beyond the VaR threshold

7. VaR Using the Parametric Method

Parametric VaR is a method of calculating Value at Risk assuming that the returns follow a specific probability distribution. In the standard variation, a normal distribution is assumed, but one can use other distributions as well, as we will see in the second example where we assume a t-distribution.

7.1 Normal Distribution Parametric VaR

In the case of a normal distribution, the use of a Z-table is essential in calculating the VaR. While performing the task, we need to always take into account that the risk refers to the **left tail**.

$$\text{VaR} = \mu + \sigma Z_\alpha$$

where

- μ is the mean return
- σ is the standard deviation of returns
- Z_α is the Z-score that corresponds to the confidence interval

Considerations for this method:

- If returns are not normally distributed, you will likely be underestimating the true Value at Risk. For example, many stocks have more outliers than the normal distribution would assume; this means the computed Value at Risk will be lower than what it is in actuality.
- Variance and covariance between return streams must be considered when computing VaR for a portfolio. Even if returns are normally distributed, the VaR calculation can still be thrown off if the estimated variances and covariances are incorrect. This can be further amplified if variances and covariances are changing over time.
- Models that allow variance to change over time (heteroskedasticity) display a greater degree of accuracy. Engle has argued that autoregressive conditional heteroskedasticity (ARCH) and generalized autoregressive conditional heteroskedasticity (GARCH) models provide better forecasts of variance and, by extension, better measures of Value at Risk.
- This method breaks down whenever a portfolio has assets with a non-linear payoff structure, e.g., options.

7.2 Implementing Parametric VaR with Normal Distribution

While it's well-established that financial returns, particularly for equities, often exhibit non-normal characteristics, such as skewness and fat tails, this parametric approach still holds significant value in financial risk management. The normal distribution assumption provides an efficient way to estimate risk, making it highly useful in situations where quick approximations are needed or can be used in conjunction with other models.

To calculate Value at Risk (VaR) using the parametric method, we will leverage the `norm.ppf()` function from Python's `scipy.stats` module. The percent point function (ppf) is actually the inverse cumulative distribution function (CDF), and it allows us to compute the quantile that corresponds to a given confidence level.

```
]: mean = btcusd_returns.mean()
std = btcusd_returns.std()

var_90 = stats.norm.ppf(0.1, mean, std)[0]
var_95 = stats.norm.ppf(0.05, mean, std)[0]
var_99 = stats.norm.ppf(0.01, mean, std)[0]

print(f"The parametric VaR assuming a normal distribution with a 90% confidence interval, is {round(100 * var_90, 3)} \n")
print(f"The parametric VaR assuming a normal distribution with a 95% confidence interval, is {round(100 * var_95, 3)} \n")
print(f"The parametric VaR assuming a normal distribution with a 99% confidence interval, is {round(100 * var_99, 3)} \n")
```

```
The parametric VaR assuming a normal distribution with a 90% confidence interval, is -4.716

The parametric VaR assuming a normal distribution with a 95% confidence interval, is -6.108

The parametric VaR assuming a normal distribution with a 99% confidence interval, is -8.72
```

Excess kurtosis observed in asset return distributions indicates that the normal distribution is not the best choice to represent the returns, especially in the tails. Their fat-tailed nature indicates a higher probability of extreme outcomes, in many cases a lot higher than what a normal distribution predicts. The latter forces us to employ a heavy-tailed distribution like the Student's t-distribution.

7.3 Implementing Parametric VaR with t-distribution

The Student's t-distribution is commonly used as an alternative to the normal distribution when modeling data with fat tails. The degrees of freedom parameter controls the heaviness of the tails. As the degrees of freedom increase, the t-distribution approaches the normal distribution. For smaller degrees of freedom (dof), the tails become heavier, indicating a higher probability of extreme values.

```
]: # degrees of freedom

def getTVar(returns, dof, confidenceLevel):
    mean = returns.mean()
    std = returns.std()
    var = np.sqrt((dof - 2) / dof) * stats.t.ppf(1 - confidenceLevel, dof) * std + mean
    return (100 * var).round(3)

print(f"The parametric VaR using t-distribution with a confidence interval 0.9, is {getTVar(btcusd_returns, 5, 0.9)}% \n")
print(f"The parametric VaR using t-distribution with a confidence interval 0.95, is {getTVar(btcusd_returns, 5, 0.95)}% \n")
print(f"The parametric VaR using t-distribution with a confidence interval 0.99, is {getTVar(btcusd_returns, 5, 0.99)}% \n")
```

```
The parametric VaR using t-distribution with a confidence interval 0.9, is Ticker
BTC-USD   -4.186
dtype: float64%

The parametric VaR using t-distribution with a confidence interval 0.95, is Ticker
BTC-USD   -5.786
dtype: float64%

The parametric VaR using t-distribution with a confidence interval 0.99, is Ticker
BTC-USD   -9.793
dtype: float64%
```

8. VaR Using Monte Carlo Simulation

In the previous sections, we discussed how one can use historical data on financial series to compute the Value at Risk. But what if we have incomplete or insufficient data to do so? In this case, we can hypothesize a distribution that generates the data and run simulations using those distributional assumptions. This method is called *Monte Carlo VaR*.

The Monte Carlo VaR includes a series of simulations, where each return stream is represented as a random variable. This variable can be taken from any probability distribution, which is great because that means it doesn't necessarily assume a normal distribution. There is a lot of flexibility in choosing what kind of distribution to use. All the variables are then dollar-weighted and simulated to see what the total portfolio value is at the end of each run. These simulation returns are then sorted lowest to highest, and we can easily look to see what the Value at Risk is using similar computations to the historical method, except this time, we're using simulated returns instead of historical returns. For example, if you ran a series of 1,000 simulations, you would look at the 50th lowest value to determine the VaR for a 95% confidence interval.

Considerations for this method:

- Estimations will not be effective if the probability distributions used to determine the random variables are incorrect. Many use past data to get an idea of what the probability distribution should be; this method allows for some subjectivity.
- You can estimate VaR more effectively for portfolios containing options with this method versus the parametric method since Monte Carlo doesn't assume a normal distribution of returns.

hypothesize a distribution that generates the data and run simulations using those distributional assumptions. This method is called Monte Carlo VaR

when to use: have incomplete or insufficient data to compute the Value at Risk

each return stream is represented as a random variable. This variable can be taken from any probability distribution.

There is a lot of flexibility in choosing what kind of distribution to use. All the variables are then dollar-weighted and simulated to see what the total portfolio value is at the end of each run.

These simulation returns are then sorted lowest to highest.

Note:

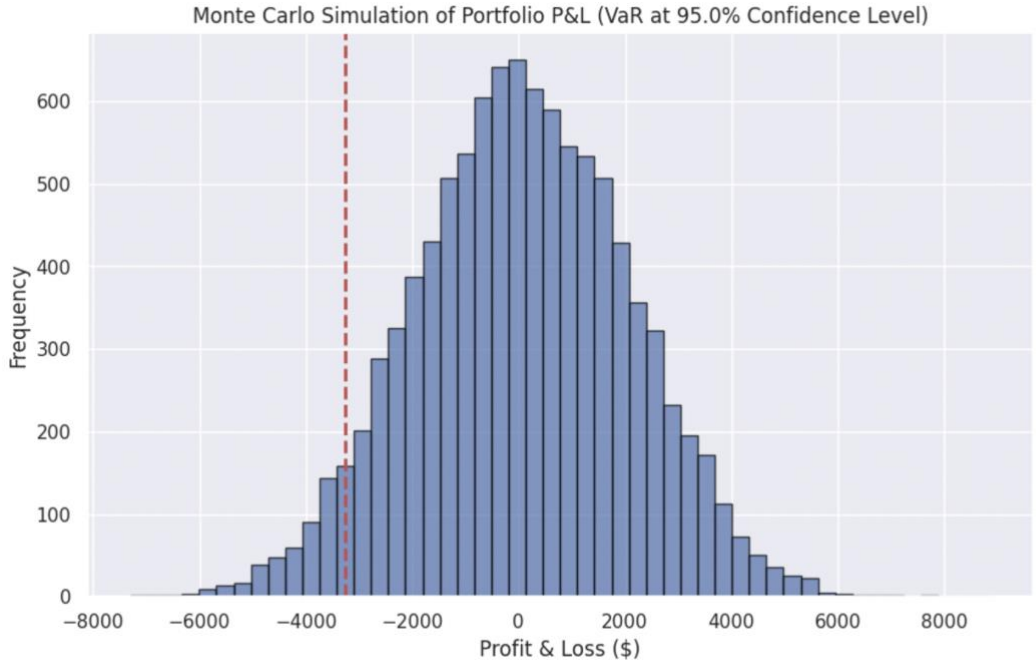
1. Estimations will not be effective if the probability distributions used to determine the random variables are incorrect.

2. You can estimate VaR more effectively for portfolios containing options with this method versus the parametric method since Monte Carlo doesn't assume a normal distribution of returns

```python
# Parameters
initial_investment = 100000  # Initial portfolio value (in dollars)
mean_return = 0.0005  # Daily mean return of the asset (e.g., 0.05%)
volatility = 0.02  # Daily volatility (not annual!) is 2%
days = 1  # 1 day time horizon
num_simulations = 10000  # Number of Monte Carlo simulations
confidence_level = 0.95  # Confidence level for VaR (e.g., 95%)

# Generate random returns based on normal distribution
np.random.seed(2024)  # For reproducibility
random_returns = np.random.normal(loc=mean_return * days, scale=volatility * np.sqrt(days), size=num_simulations)

# Calculate portfolio values at the end of the simulation
portfolio_end_values = initial_investment * (1 + random_returns)

# Calculate profit and loss (P&L)
portfolio_pnl = portfolio_end_values - initial_investment

# Calculate the VaR at the specified confidence level
VaR = np.percentile(portfolio_pnl, (1 - confidence_level) * 100)

# Output the results
print(f"Value at Risk (VaR) at {confidence_level * 100}% confidence level: -${abs(VaR):,.2f}")

#Plot the distribution of portfolio P&L
plt.figure(figsize=(10, 6))
plt.hist(portfolio_pnl, bins=50, edgecolor='black', alpha=0.7)
plt.axvline(x=VaR, color='r', linestyle='dashed', linewidth=2)
plt.title(f"Monte Carlo Simulation of Portfolio P&L (VaR at {confidence_level * 100}% Confidence Level)")
plt.xlabel('Profit & Loss ($)')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

Generate random returns based on normal distribution

```
Value at Risk (VaR) at 95.0% confidence level: -$3,253.02
```



9. Conclusion

In this lesson, we've introduced a wide range of important concepts in financial risk analysis. We started by examining different types of returns, including total returns, dividend-adjusted returns, and excess returns, which form the foundation for more advanced risk metrics. We then explored the Sharpe Ratio and showcased how important this ratio is in the portfolio management context.

The latter part of the lesson focused on downside risk metrics, particularly Value at Risk (VaR). We covered various methods for calculating VaR, including historical, parametric (with both normal and t-distributions), and Monte Carlo simulation approaches. We also introduced the concept of Conditional Value at Risk (CVaR) as an extension of VaR.

L3: Tick Data

```
# Load the data, dataset (1) - BTC/USD from Kraken
btc_usdt = pd.read_csv("XBTUSDT.csv", names=['UNIX_time', 'price', 'volume'])
btc_usdt.head()
```

	UNIX_time	price	volume
0	1688169604	30474.1	0.018800
1	1688169604	30474.8	0.000103
2	1688169604	30476.5	0.014997
3	1688169656	30483.5	0.000614
4	1688170008	30479.0	0.000328

The dataset above is an example of a tick-by-tick trades dataset, and it consists of all the trades that occurred in a specified interval. The first thing that you notice is the UNIX_time column, which does resemble time or date! This timestamp is called "UNIX time," and it counts the number of time units (seconds, milliseconds, microseconds, and nanoseconds) from 00:00:00 UTC January 1, 1970, until now.

This time representation provides:

- Standardization: universal representation of time.
- Simplicity: it is a single integer.
- Efficiency: calculations are straightforward and computationally efficient.

The Unix Timestamps can be represented with different levels of precision:

- 10 digits: Represents seconds (e.g., 1688169604).
- 13 digits: Includes milliseconds (e.g., 1688169604123).
- 16 digits: Includes microseconds (e.g., 1688169604123456).

The Unix Timestamps can be represented with different levels of precision:

- 10 digits: Represents seconds (e.g., 1688169604).
- 13 digits: Includes milliseconds (e.g., 1688169604123).
- 16 digits: Includes microseconds (e.g., 1688169604123456).
- 19 digits: Includes nanoseconds (e.g., 1688169604123456789).

You can also check this website.

In the dataset (1), we can count 10 digits for the UNIX_time column. Therefore, the precision is given in seconds, and this is the only thing we need to know in order to transform Unix time into human-readable datetime.

[3]:
```
# Convert the UNIX timestamp to human readable datetime. Check the argument unit='s' to make sure the timestamp is in seconds.
btc_usdt['timestamp'] = pd.to_datetime(btc_usdt['UNIX_time'], unit='s')
btc_usdt.set_index('timestamp', inplace=True) # Set the timestamp as the index
btc_usdt.head()
```

[3]:

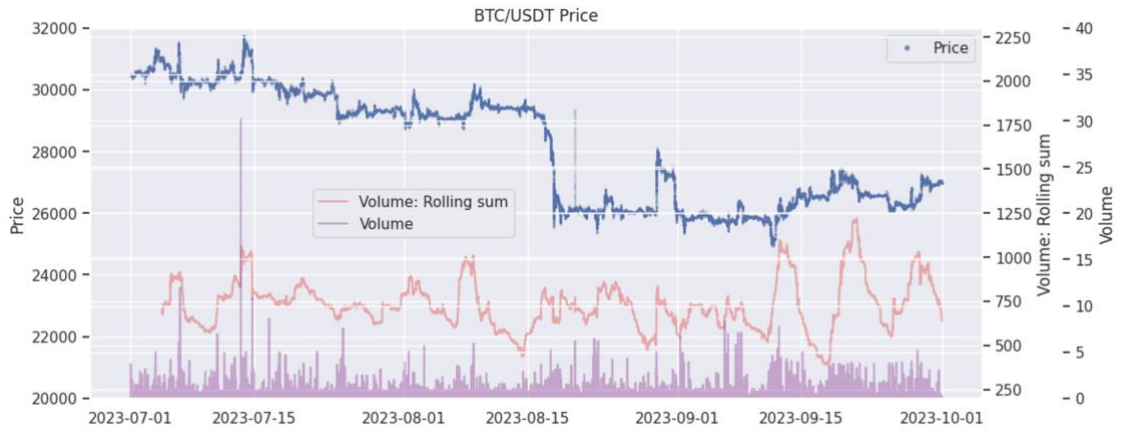
timestamp	UNIX_time	price	volume
2023-07-01 00:00:04	1688169604	30474.1	0.018800
2023-07-01 00:00:04	1688169604	30474.8	0.000103
2023-07-01 00:00:04	1688169604	30476.5	0.014997
2023-07-01 00:00:56	1688169656	30483.5	0.000614
2023-07-01 00:06:48	1688170008	30479.0	0.000328

Visualizing tick trade data:

3. Visualizing Tick Trade Data

Now, let's examine our dataset visually. Visualizing tick data presents unique challenges due to its extremely high resolution. Consider this analogy: imagine you have an ultra-high-resolution image displayed on a small screen or viewed from a distance. Much of the detail is lost, and you might wonder about the value of such high resolution. Similarly, when plotting tick data, we risk obscuring important patterns if we simply plot every point. The key is to find ways to represent this high-frequency data that reveal its inherent patterns and characteristics without overwhelming the viewer. In this section, our data is sourced from Gebbie and Nonyane's "TRTH JSE AGLJ.J Intraday Transaction Test Data" and from the datasets mentioned in section 2.

```
#Figure 1
fig, ax = plt.subplots(figsize=(12, 5))
ax.scatter(btc_usdt.index, btc_usdt['price'], s = 0.001, alpha=0.4)
ax.set_ylabel('Price')
ax.set_title('BTC/USDT Price')
ax.set_ylim(20000, 32000)

custom_legend = plt.Line2D([0], [0], linestyle="none", marker='o', color='b', markersize=3, alpha = 0.7, label='Price') # Creating a custom legend
ax.legend(handles=[custom_legend], loc='upper right')

ax2 = ax.twinx() # Create a second y-axis
ax2.plot(btc_usdt.index, btc_usdt['volume'].rolling(10000).sum(), color='red', label='Volume: Rolling sum', alpha=0.3)
ax2.set_ylabel('Volume: Rolling sum')
ax2.set_ylim(200, 2300)

ax3 = ax.twinx() # Create a third y-axis
ax3.plot(btc_usdt.index, btc_usdt['volume'], color='purple', label='Volume', alpha=0.3)
ax3.set_ylabel('Volume')
ax3.set_ylim(0, 40)
ax3.spines['right'].set_position(('outward', 60))

fig.legend(bbox_to_anchor=(0.5,0.56)) # Adjust the legend position
plt.show()
```



[9]:
```
between_trades_interval = btc_usdt['UNIX_time'].diff().dropna() # Calculate the trade duration
print("Summary Statistics \n\n", between_trades_interval.describe(), "\n\n") # Get the summary statistics of the trade duration
print("10 Largest trade durations \n\n", between_trades_interval.sort_values(ascending=False).head(10)) # Get the top 10 largest durations between trades
```

```
Summary Statistics

 count    358309.000000
mean         22.184026
std          55.319270
min           0.000000
25%           0.000000
50%           0.000000
75%          20.000000
max        2195.000000
Name: UNIX_time, dtype: float64


10 Largest trade durations

 timestamp
2023-08-01 15:41:45    2195.0
2023-07-12 09:19:13    2001.0
2023-08-17 10:30:32    1809.0
2023-08-18 15:05:14    1576.0
2023-07-23 07:29:23    1281.0
2023-07-23 02:13:51    1244.0
2023-08-06 03:40:46    1224.0
2023-09-28 15:28:56    1177.0
2023-07-20 23:44:23    1125.0
2023-07-16 03:52:08    1113.0
Name: UNIX_time, dtype: float64
```

10]:
```
#Figure 2
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 4))
ax1.hist(between_trades_interval, bins=100, color='skyblue', edgecolor='black', linewidth=1.2)
ax1.set_title("Time between trades")
ax2.hist(btc_usdt['price'].pct_change().dropna(), bins=100, color='white', edgecolor='black', linewidth=1.2)
ax2.set_title("Simple Returns")
ax3.hist(btc_usdt['volume'], bins=100, color='green', edgecolor='black', linewidth=1.2)
ax3.set_title("Volume")
plt.show()
```

The above results hint at three important facts:

- There are several trades that happened at the same second. Those trades could be a single trade that was divided in many trades at the moment of execution as the volume at the best bid or ask was not enough to cover the trade volume in that instant. These trades are called **split transactions**.
- There are several trades that did not result in a price change.
- A lot of trades seem to have very low volume.

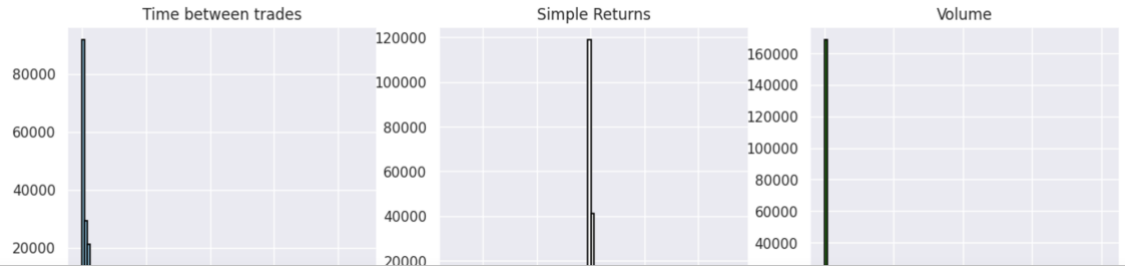
14]:
```
boolean_array_is_duplicate = btc_usdt['UNIX_time'].duplicated(keep=False) # Check for duplicates
sum_of_duplicates = boolean_array_is_duplicate.sum() # Sum the duplicates
print("The number of trades that occured at the same second with at least one more trade, is: ",sum_of_duplicates)
print("The percentage of these trades to the number of all trades, is: ",round(sum_of_duplicates / len(btc_usdt) * 100, 2), "%")
print("The percentage of the volume of these trades to the entire volume, is: ",round(((boolean_array_is_duplicate * btc_usdt['volume']).sum() / btc_usdt['volume'].sum
```

```
The number of trades that occured at the same second with at least one more trade, is:  241814
The percentage of these trades to the number of all trades, is:  67.49 %
The percentage of the volume of these trades to the entire volume, is:  90.44 %
```

We will assume that these trades are split transactions. After some simple aggregations, the histograms plotted above look very different:

We will assume that these trades are split transactions. After some simple aggregations, the histograms plotted above look very different:

[42]:
```
# Figure 3
aggr_dataset_1 = pd.DataFrame()
aggr_dataset_1['UNIX_time'] = btc_usdt['UNIX_time'].groupby(btc_usdt['UNIX_time']).first() # Group the UNIX timestamp by the first timestamp
aggr_dataset_1['volume'] = btc_usdt['volume'].groupby(btc_usdt['UNIX_time']).sum() # Aggregate the volume by the UNIX timestamp
aggr_dataset_1['price'] = (btc_usdt['price'] * btc_usdt['volume']).groupby(btc_usdt['UNIX_time']).sum() / aggr_dataset_1['volume'] # Final price as VWAP of the spli
aggr_dataset_1.index = btc_usdt.index.drop_duplicates() # Drop the duplicates from the index

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 4))
ax1.hist(aggr_dataset_1['UNIX_time'].diff().dropna(), bins=100, color='skyblue', edgecolor='black', linewidth=1.2)
ax1.set_title("Time between trades")
ax2.hist(aggr_dataset_1['price'].pct_change().dropna(),bins=100, color='white', edgecolor='black', linewidth=1.2)
ax2.set_title("Simple Returns")
ax3.hist(aggr_dataset_1['volume'], bins=100, color='green', edgecolor='black', linewidth=1.2)
ax3.set_title("Volume")
plt.show()
aggr_dataset_1
```



4. Including the Trade Direction

Let's check dataset (2), which holds the RVN/USDT trades but with one addition to the dataset (1): there is an additional column, "Trade Direction," which holds information on whether the trader who initiated the trade was a buyer or a seller. In our case, the exchange is storing a boolean value (True, False) for each row that answers the question *is the buyer the market maker?*

As we know, for each transaction, there must be a buyer and a seller. If the answer to the above question is true, then the market maker placed a bid order in the orderbook with the intention to buy, and a seller initiated the trade (partially?) filling that buying order. If, on the other hand, the answer is false, then the market maker was a seller who placed an ask order. In this case, the initiator of the trade is a buyer who filled that specific ask order.

This additional boolean array is a source of valuable information.

We will begin by loading dataset (2). In many cases, the data vendor/exchange will let you download the data in `csv` files with each file containing the data of one day (or week, or month, or custom interval). In our case, dataset (2) consists of four daily files. We will be using the `glob` library due to its simplicity in selecting the files we need: https://builtin.com/software-engineering-perspectives/glob-in-python.

]:
```
import glob

# Load the data, dataset (2) - RVN/USDT
list_of_files = glob.glob('RVNUSDT*') # List the RVNUSDT files in the directory
column_names = ['id', 'price', 'qty', 'base_qty', 'UNIX_time', 'is_buyer_maker', 'is_best_match'] # Column names
rvn_usdt = pd.concat([pd.read_csv(file, names=column_names) for file in list_of_files], ignore_index=True, axis = 0) # Concatenate the files
rvn_usdt.sort_values('UNIX_time', inplace=True) # Sort the values by time
rvn_usdt.head(5)
```

]:

	id	price	qty	base_qty	UNIX_time	is_buyer_maker	is_best_match
964617	54407520	0.02588	424.9	10.996412	1680307215745	False	True
964618	54407521	0.02588	3313.5	85.753380	1680307215759	False	True
964619	54407522	0.02588	4172.5	107.984300	1680307215759	False	True
964620	54407523	0.02588	5162.5	133.605500	1680307215762	False	True

The `base_qty` represents the volume of the base currency (the currency that the asset is traded against) traded given the volume of the asset traded and the price of the trade $\text{base\_qty} = \text{qty} * \text{price}$. The column `id` represents the id of each trade and can be used to filter out duplicate entries. Lastly, the column `is_best_match` is a boolean array that indicates whether the trade was executed at the best possible price given the orderbook at that time (True or False).

We will check whether the additional columns hold any information before we omit them in order to handle a smaller dataset, which will be less taxing on ram.

[12]:
```
print("Number of duplicate entries: ", rvn_usdt['id'].duplicated().sum()) # Check for duplicates
print("Number of entries that were not executed at the best possible price: ", (~rvn_usdt['is_best_match']).sum()) # Check for best match
print("Can the base_qty be derived from the qty and price?: ", np.isclose(rvn_usdt['price'] * rvn_usdt['qty'], rvn_usdt['base_qty'], atol=1e-05).all()) # Check if the
print("Due to the results above, we can drop the columns 'id', 'is_best_match' and 'base_qty' \n\n")

rvn_usdt = rvn_usdt.drop(columns=['id', 'is_best_match', 'base_qty']) # Drop the unnecessary columns
rvn_usdt['timestamp'] = pd.to_datetime(rvn_usdt['UNIX_time'], unit='ms') # Convert the UNIX timestamp to human readable datetime. Check the argument unit='ms' to make
rvn_usdt.set_index('timestamp', inplace=True) # Set the timestamp as the index
rvn_usdt.head()
```

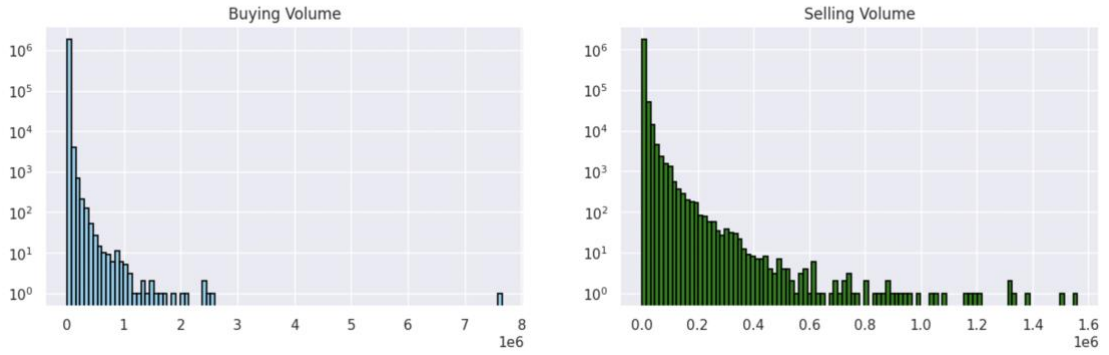
```
Number of duplicate entries:  0
Number of entries that were not executed at the best possible price:  0
Can the base_qty be derived from the qty and price?:  True
Due to the results above, we can drop the columns 'id', 'is_best_match' and 'base_qty'
```

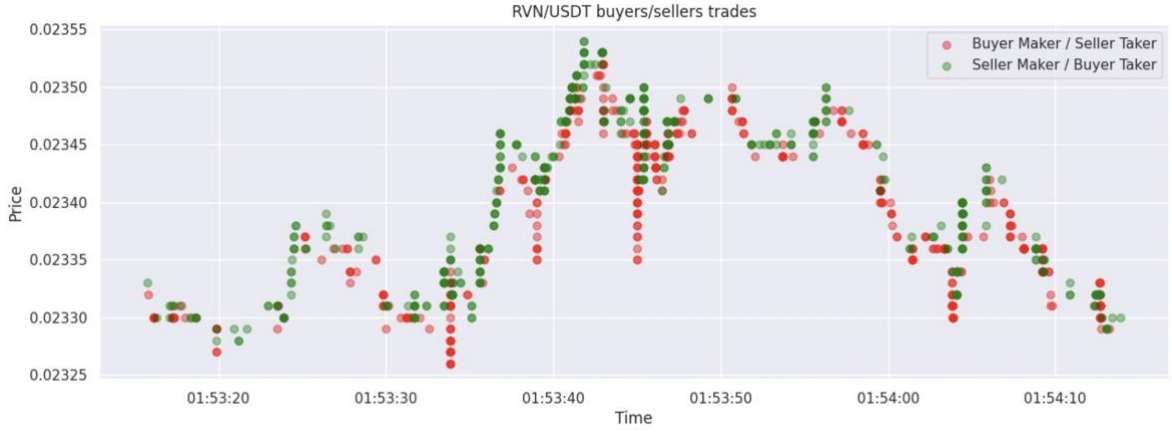
[12]:

timestamp	price	qty	UNIX_time	is_buyer_maker
2023-04-01 00:00:15.745	0.02588	424.9	1680307215745	False
2023-04-01 00:00:15.759	0.02588	3313.5	1680307215759	False
2023-04-01 00:00:15.759	0.02588	4172.5	1680307215759	False
2023-04-01 00:00:15.762	0.02588	5162.5	1680307215762	False
2023-04-01 00:00:15.762	0.02588	1041.3	1680307215762	False

```
print("The buying volume is: ", buying_volume.sum()) # Calculate the buying volume
print("The selling volume is: ", selling_volume.sum()) # Calculate the selling volume

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 4))
ax1.hist(buying_volume, log = True, bins=100, color='skyblue', edgecolor='black', linewidth=1.2)
ax1.set_title("Buying Volume")
ax2.hist(selling_volume, log = True, bins=100, color='green', edgecolor='black', linewidth=1.2)
ax2.set_title("Selling Volume")
plt.show()
buying_volume
```

```
The buying volume is:  6081743642.6
The selling volume is:  6530522239.799997
```





6. Level 1 Data (L1)

Level 1 data (or else top-of-book data (TOB)) typically include the following columns:

- Symbol: the unique identifier of the financial instrument.
- Timestamp: the exact time when the event was recorded.
- Bid price: the highest price a buyer is willing to pay.
- Ask price: the lowest price a seller is willing to accept.
- Ask volume: number of units available at the ask price.
- Bid Volume: number of units available at the bid price.

[23]:
```
# Load the data, dataset (3) - AGLJ_J
columns = ['RIC', 'DateTimeL', 'Type', 'Price', 'Volume', 'L1 Bid', 'L1 Ask', 'Trade Sign']
aglj = pd.read_csv('AGLJ_J_04-Jan-2016_TO_10-May-2016_5days.csv', \
                   names = columns, skiprows = 1) # Load the data
aglj.sort_values(by = 'DateTimeL', inplace = True)
aglj.head(5)
```

[23]:

	RIC	DateTimeL	Type	Price	Volume	L1 Bid	L1 Ask	Trade Sign
1	AGLJ.J	736333.382013	Quote	0.0	0.0	0.0	6800.0	0
0	AGLJ.J	736333.382013	Trade	6750.0	366.0	0.0	0.0	0
2	AGLJ.J	736333.382015	Trade	6750.0	374.0	0.0	0.0	0
4	AGLJ.J	736333.382015	Quote	0.0	0.0	6735.0	6800.0	0
3	AGLJ.J	736333.382015	Quote	0.0	0.0	0.0	6800.0	0

Change date:

;]:

timestamp	RIC	DateTimeL	Type	Price	Volume	L1 Bid	L1 Ask	Trade Sign
2016-01-04 09:10:05	AGLJ.J	736333.382013	Quote	0.0	0.0	0.0	6800.0	0
2016-01-04 09:10:05	AGLJ.J	736333.382013	Trade	6750.0	366.0	0.0	0.0	0
2016-01-04 09:10:06	AGLJ.J	736333.382015	Trade	6750.0	374.0	0.0	0.0	0
2016-01-04 09:10:06	AGLJ.J	736333.382015	Quote	0.0	0.0	6735.0	6800.0	0
2016-01-04 09:10:06	AGLJ.J	736333.382015	Quote	0.0	0.0	0.0	6800.0	0

Let's now apply some cleaning filters:

Before we can effectively analyze high-frequency financial data, it's crucial to clean and preprocess the dataset. This cleaning process helps to remove erroneous entries and adjust for certain market microstructure effects, ensuring that our subsequent analyses are based on accurate and meaningful data.

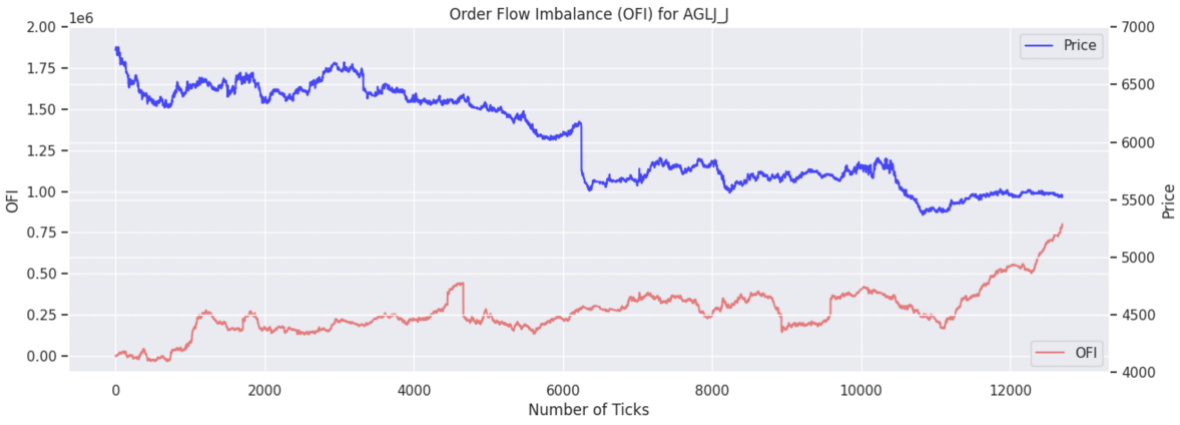
First, we remove entries with zero or negative prices, quotes, or volumes. These are likely data errors, as prices and volumes in financial markets should always be positive. Negative spreads (where the bid price exceeds the ask price) are also eliminated as they represent impossible market conditions under normal circumstances.

Next, we address the issue of split transactions. In high-frequency trading, a single large order may be executed as multiple smaller trades at the same timestamp. This can lead to an overrepresentation of certain price levels and potentially skew our analysis. To mitigate this, we aggregate trades occurring at the same timestamp, summing their volumes and calculating a volume-weighted average price (VWAP). This approach preserves overall trading information while avoiding the artificial inflation of trade counts. Similarly, we deal with intra-second quotes by keeping only the last quote for each timestamp. This is because, in a fast-moving market, the most recent quote is typically the most relevant for analysis.

7. Order Flow Imbalance (OFI)

Order Flow Imbalance (OFI) is an important concept in high-frequency data analysis that provide insights into the buying and selling pressures in the market. It measures the net order flow by taking into account the direction and size of trades. We will calculate the simplest form of OFI by multiplying the `Trade Sign` column (a positive sign for buyer-initiated trades and a negative sign for seller-initiated trades) with the respective trade volumes. Over time, the cumulative sum of these signed volumes gives us the OFI. A positive OFI indicates net buying pressure, while a negative OFI suggests net selling pressure. By analyzing OFI, we can gain valuable insights into short-term price movements, liquidity dynamics, and potential future price trends (as theory suggests).

53]:
```
# Figure 6
signed_volume = aggr_dataset_2[aggr_dataset_2['Type'] == ' Trade']['Volume'] * aggr_dataset_2[aggr_dataset_2['Type'] == ' Trade']['Trade Sign'] # Calculate the OFI
ofi = signed_volume.cumsum() # Calculate the cumulative sum of the OFI
prices = aggr_dataset_2[aggr_dataset_2['Type'] == ' Trade']['Price'] # Get the prices

fig, ax = plt.subplots(figsize=(15, 5))
ax.plot(np.arange(len(ofi)), ofi, label = 'OFI', color = 'red', alpha = 0.5) # Plot the OFI
ax.set_ylabel('OFI')
ax.set_xlabel('Number of Ticks')
ax.set_title('Order Flow Imbalance (OFI) for AGLJ_J')
ax.set_ylim(-100000, 2000000)
ax.legend(loc = 'lower right')

ax2 = ax.twinx()
ax2.plot(np.arange(len(prices)), prices, color='blue', label='Price', alpha=0.7)
ax2.set_ylabel('Price')
ax2.legend()
ax2.set_ylim(4000, 7000)
plt.show()

print("\n\nThe Pearson correlation between the price and OFI is: ", np.corrcoef(prices, ofi)[0, 1]) # Calculate the Pearson correlation between the price and OFI
```



Exercise 5

Import the library `scipy` and compute the Spearman and Kendall correlation between the price and OFI. What do you see? Was that result what you expected?

The above result is somewhat counterintuitive and very common in practice. It highlights the importance of empirical analysis with real-world data rather than relying on only theoretical expectations.

As we can see, the price of AGLJ during these five days has a downward slope. On several occasions, the price has a short-term upward slope, but the trend in general is clear. Given the price, our expectation (based on theory and intuition) would be a downward slope for the OFI as well. But this is hardly the case since the OFI seems almost completely inconsistent with the price. The buying volume (as defined in this notebook) is rising which means that there are more buyers (takers) than sellers whereas, at the same time, the price is falling.

There could be several reasons for this phenomenon that could be very different if the data were retrieved from another exchange or for some other time interval:

- Market maker/Trader behavior: Market makers or large traders might be providing liquidity as prices fall, absorbing sell orders. This would show up as positive OFI (as they're buying) even as prices continue to decline.
- Large trades: A few large selling orders could outweigh the importance of many small buying orders.
- Market Microstructure Effects: In some market structures, large sell orders can trigger a series of smaller buy orders as they're filled, which could lead to this pattern.
- Time Frame Consideration: The relationship between OFI and price might be different over different time scales. What we're seeing could be a short-term effect that might reverse over longer periods.

a definition for buying and selling volume: Buying/Selling volume is the volume of the trades that were initiated by buyers/sellers (as takers)

L4: singular value decomposition of Matrices

1. Introducing Singular Value Decomposition (SVD)

Singular value decomposition (SVD) is a very popular method in numerical linear algebra. It is commonly used to solve for a linear regression problem when the matrix is not squared. It can also be used as a data dimension reduction method. One of the examples is to use SVD as the basis for principal component analysis. SVD is also a foundational numeric method for machine learning algorithms.
In the last module, we discussed how to diagonalize a symmetric matrix. If A is a 3 by 3 symmetric matrix, we can diagonalize or factor A as follows:

$$A = \begin{bmatrix} | & | & | \\ v_1 & v_2 & v_3 \\ | & | & | \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \begin{bmatrix} | & | & | \\ v_1 & v_2 & v_3 \\ | & | & | \end{bmatrix}^T$$

where
v_1, v_2, v_3 are eigenvectors of A
$\lambda_1, \lambda_2, \lambda_3$ are eigenvalues of A

However, most of the time, the datasets or a matrix we have will not have symmetry. Hence, we need a method to diagonalize a general matrix. This method is singular value decomposition.