Financial Risk Forecasting
Seminar Week 2: Data Download and Visualisation

Jon Danielsson
London School of Economics

Version 4.0

2 Data analysis

2.1 Why data download and visualisation matter for financial risk

Accurate financial data is the foundation of all risk management decisions. The ability to systematically download, clean and visualise market data enables risk managers to identify patterns, outliers and trends that drive portfolio risk. Modern risk systems require automated data pipelines that can handle multiple assets and time periods efficiently.

Visualisation skills are essential for communicating risk insights to stakeholders and identifying data quality issues before they impact risk calculations.

For more detail, see Loading Data and Plots in the R notebook.

2.2 The plan for this week

1. Install and load additional packages
2. Understand data frames and matrices for financial data
3. Download and explore individual stock data
4. Calculate returns and handle data alignment issues
5. Create reusable functions for data processing
6. Scale to multiple stocks efficiently
7. Create structured data frames for prices and returns
8. Visualise and analyse the data
9. Save and export data and plots

2.3 Libraries / packages

One of the most useful features of R is its large library of packages. Although the base R language is very powerful, there are thousands of community-built packages to perform specific tasks that can save us the effort of programming some things from scratch. The complete list of available R packages can be found on CRAN.

To install a package in RStudio, go to the `Tools` menu and the first item is `Install Packages`. You can also type `install.packages("name_of_package", repos = "https://cloud.r-project.org/")` in the console. After installation, you can load a package/library with: `library(name_of_package)`.

It is a good practice to start your R script by loading all the libraries that will be used in your program.

2.4 Loading packages needed for today

```r
library(eodhdR2)
library(lubridate)
```

```

Attaching package: 'lubridate'

The following objects are masked from 'package:base':

    date, intersect, setdiff, union
```

2.5 Data frames and matrices for financial data

Data frames and matrices are fundamental data structures in R for handling financial data. Data frames can hold different data types (numeric, character, dates) in columns, making them ideal for stock data with prices, dates and ticker symbols. Matrices are more efficient for numerical operations but can only contain one data type, making them perfect for price or return calculations across multiple assets.

Data types specify what kind of information a variable can store — for example, numbers (numeric), text (character) or dates. See Variables in the R Notebook for more detail on R data types.

See Data frames and matrices in the R Notebook.

2.6 Finding stock symbols on EOD Historical Data

Before we start downloading data, it is important to know how to find stock symbols, see Financial Data in the R Notebook. You can visit eodhd.com/ to explore available stocks.

1. Use the search bar to find specific companies
2. Browse by exchange (e.g., US, LSE, TSE)
3. View the symbol format: `TICKER.EXCHANGE` (e.g., `AAPL.US`)

For US stocks, the `.US` extension is required when using the API. The demo token provides access to a limited set of symbols, including AAPL.US, TSLA.US, VTI.US, AMZN.US, BTC-USD.CC and EURUSD.FOREX. For indices, the extension is `indx`.

2.6.1 Securities we analyse in this seminar

Individual Stocks:

- AAPL — Apple Inc. (Technology)
- MSFT — Microsoft Corporation (Technology)
- NVDA — Nvidia Corporation (Technology/Semiconductors)
- JPM — JPMorgan Chase & Co. (Financial Services)
- C — Citigroup Inc. (Financial Services)
- XOM — Exxon Mobil Corporation (Energy)
- MCD — McDonald's Corporation (Consumer Discretionary)
- GE — General Electric Company (Industrial)

Market Indices:

- GSPC — S&P 500 Index
- IXIC — NASDAQ Composite Index

2.7 Setting up and downloading data

First, set up our API token and define the stocks we want to analyse. All of these symbols have been verified as available on the US exchange:

```
# Set up your API token
set_token(token)
```

Define all securities we want to analyse. These are all valid symbols on EOD Historical Data

```
tickers = c("GSPC","IXIC","AAPL", "MSFT", "JPM", "C", "XOM", "MCD", "GE",  "NVDA")

# Note: US Stocks use US extension, indices use INDX
exchanges = c(rep("INDX", 2),rep("US", 9))

# Create a mapping for company/index names
security_names = c(
  "GSPC" = "S&P 500",
  "IXIC" = "NASDAQ Composite",
  "AAPL" = "Apple",
  "MSFT" = "Microsoft",
  "JPM" = "JP Morgan",
  "C" = "Citigroup",
  "XOM" = "Exxon",
  "MCD" = "McDonald's",
  "GE" = "General Electric",
  "NVDA" = "Nvidia"
)
```

2.8 Working with individual stocks first

Before scaling to multiple stocks, understand the complete workflow with a single stock. This approach helps identify the steps we will need to automate later.

2.8.1 Downloading individual stock data

```r
# Download Apple stock data
apple_data = get_prices("AAPL", "US")
```

```
-- retrieving price data for ticker AAPL|US ------------------------------------

! Quota status: 75|100000, refreshing in 15.3 hours

i cache file AAPL_US_eodhd_prices.rds saved

v got 11255 rows of prices

i got daily data from 1980-12-12 to 2025-08-08
```

```r
# Examine the structure
head(apple_data)
```

```
        date    open    high     low   close adjusted_close    volume ticker
1 1980-12-12 28.7392 28.8736 28.7392 28.7392         0.0986 469033600   AAPL
2 1980-12-15 27.3728 27.3728 27.2608 27.2608         0.0935 175884800   AAPL
3 1980-12-16 25.3792 25.3792 25.2448 25.2448         0.0866 105728000   AAPL
4 1980-12-17 25.8720 26.0064 25.8720 25.8720         0.0887  86441600   AAPL
5 1980-12-18 26.6336 26.7456 26.6336 26.6336         0.0914  73449600   AAPL
6 1980-12-19 28.2464 28.3808 28.2464 28.2464         0.0969  48630400   AAPL
  exchange ret_adj_close
1       US            NA
2       US   -0.05172414
3       US   -0.07379679
4       US    0.02424942
5       US    0.03043968
6       US    0.06017505
```

```r
names(apple_data)
```

```
 [1] "date"           "open"           "high"           "low"           
 [5] "close"          "adjusted_close" "volume"         "ticker"        
 [9] "exchange"       "ret_adj_close" 
```

```r
dim(apple_data)
```

```
[1] 11255    10
```

2.8.2 Adding returns to individual stock data

Recall from Week 1 how we calculated returns. Apply the same approach here:

```r
# Calculate log returns
y = diff(log(apple_data$adjusted_close))
# Check the length difference
cat("Original data has", nrow(apple_data), "rows\n")
```

```
Original data has 11255 rows
```

```r
cat("Returns vector has", length(y), "values\n")
```

```
Returns vector has 11254 values
```

We cannot directly add the returns to the data frame with the prices because of the length mismatch:

```r
# This would give an error
# apple_data$returns = y
# Error: replacement has X rows, data has X+1
```

This happens because the first return corresponds to the second day. We have two options:

2.8.3 Option 1: Pad with NA

```r
# Add NA at the beginning to match lengths
apple_data$returns = c(NA, y)
head(apple_data[, c("date", "adjusted_close", "returns")])
```

```
        date adjusted_close     returns
1 1980-12-12         0.0986          NA
2 1980-12-15         0.0935 -0.05310983
3 1980-12-16         0.0866 -0.07666162
4 1980-12-17         0.0887  0.02396007
5 1980-12-18         0.0914  0.02998559
6 1980-12-19         0.0969  0.05843404
```

2.8.4 Option 2: Remove the first row

```r
# Alternative: remove the first row entirely
apple_data_with_returns = apple_data[2:nrow(apple_data),]
apple_data_with_returns$returns = y
head(apple_data_with_returns[, c("date", "adjusted_close", "returns")])
```

```
        date adjusted_close     returns
2 1980-12-15         0.0935 -0.05310983
3 1980-12-16         0.0866 -0.07666162
4 1980-12-17         0.0887  0.02396007
5 1980-12-18         0.0914  0.02998559
6 1980-12-19         0.0969  0.05843404
7 1980-12-22         0.1016  0.04736402
```

Both approaches work — choose an approach based on your analysis needs. Option 1 keeps all dates but has NA for the first return. Option 2 has complete data but loses the first date.

2.9 Creating reusable functions

Now that we understand the individual stock workflow, we can create functions to automate repetitive tasks. This is useful when working with multiple stocks.

2.9.1 Function for calculating returns

Create a reusable function that implements Option 2 (removing the first row):

```r
# Function to add returns and remove the first row
add_returns = function(df) {
  # Calculate log returns
  returns = diff(log(df$adjusted_close))
  # Remove the first row
  df = df[2:nrow(df), ]
  # Add returns column
  df$returns = returns
  return(df)
}

# Test the function
apple_with_returns = add_returns(apple_data)
head(apple_with_returns[, c("date", "adjusted_close", "returns")])
```

```
        date adjusted_close     returns
2 1980-12-15         0.0935 -0.05310983
3 1980-12-16         0.0866 -0.07666162
4 1980-12-17         0.0887  0.02396007
5 1980-12-18         0.0914  0.02998559
6 1980-12-19         0.0969  0.05843404
7 1980-12-22         0.1016  0.04736402
```

This function makes it easy to add returns to any stock data frame consistently.

2.10 Scaling to multiple stocks

Now that we have a clear workflow and reusable function, we can efficiently process multiple securities. This demonstrates why functions are valuable — they eliminate repetitive code and ensure consistency.

2.10.1 Downloading multiple stocks with returns

```r
# Create an empty list to store data with returns
data = list()

# Download data for each ticker and add returns
for(i in 1:length(tickers)) {
  ticker = tickers[i]
  exchange = exchanges[i]
  cat("Downloading", security_names[ticker], "(", ticker, ")\n")
  # Can also do
  #cat("Downloading", security_names[tickers[i]], "(", tickers[i], ")\n")

  # Get price data
  # suppressMessages() prevents printing messages to the console
  prices = suppressMessages(get_prices(ticker, exchange))
  # or
  prices = suppressMessages(get_prices(tickers[i], exchanges[i]))


  # Add ticker column for identification
  prices$ticker = ticker
  # or
  prices$ticker = tickers[i]

  # Apply the add_returns function
  data[[ticker]] = add_returns(prices)
}
```

```
Downloading S&P 500 ( GSPC )
Downloading NASDAQ Composite ( IXIC )
Downloading Apple ( AAPL )
Downloading Microsoft ( MSFT )
Downloading JP Morgan ( JPM )
Downloading Citigroup ( C )
Downloading Exxon ( XOM )
Downloading McDonald's ( MCD )
Downloading General Electric ( GE )
Downloading Nvidia ( NVDA )
```

```
# Check the structure
cat("\nExample data with returns:\n")
```

```

Example data with returns:
```

```
head(data[["AAPL"]][, c("date", "adjusted_close", "returns")])
```

```
        date adjusted_close     returns
2 1980-12-15         0.0935 -0.05310983
3 1980-12-16         0.0866 -0.07666162
4 1980-12-17         0.0887  0.02396007
5 1980-12-18         0.0914  0.02998559
6 1980-12-19         0.0969  0.05843404
7 1980-12-22         0.1016  0.04736402
```

2.11 Organizing data for analysis

When working with multiple stocks, we must organise the data for efficient analysis. Individual stock data frames are useful for detailed analysis, but portfolio analysis and visualisation require structured data frames with aligned dates.

2.11.1 Creating structured price and return data frames

The goal is to create data frames where each column represents a different security and each row represents a date. This structure is helpful for:

- Calculating correlations between stocks

- Portfolio analysis
- Creating comparative visualisations
- Risk calculations across multiple assets

2.11.2 Getting reference dates

When creating aligned data frames, we need a common set of dates. We will explore two approaches to understand the underlying challenges.

2.11.2.1 Simple approach: assume aligned dates

The simplest approach assumes all stocks trade on the same days as a major index:

```r
# Use S&P 500 dates as reference (exclude today for data consistency)
reference_dates = data[["GSPC"]]$date
reference_dates = reference_dates[reference_dates != Sys.Date()]

cat("Using", length(reference_dates), "reference dates from S&P 500\n")
```

```
Using 24516 reference dates from S&P 500
```

```r
cat("Date range:", as.character(min(reference_dates)),
    "to", as.character(max(reference_dates)), "\n")
```

```
Date range: 1928-01-03 to 2025-08-08
```

2.11.3 Creating the data frames

2.11.3.1 Approach 1: Simple assignment (assuming perfect alignment)

```r
# Try the simple approach first
Prices_simple = data.frame(date = reference_dates)
Returns_simple = data.frame(date = reference_dates)

# Attempt to add each ticker directly
for(ticker in tickers) {
  ticker_data = data[[ticker]]

  # This will fail if dates don't align perfectly
  tryCatch({
    Prices_simple[[ticker]] = ticker_data$adjusted_close
    Returns_simple[[ticker]] = ticker_data$returns
    cat("Added", ticker, "successfully\n")
  }, error = function(e) {
    cat("Error adding", ticker, ":", conditionMessage(e), "\n")
  })
}
```

```
Added GSPC successfully
Error adding IXIC : replacement has 13742 rows, data has 24516
Error adding AAPL : replacement has 11254 rows, data has 24516
Error adding MSFT : replacement has 9928 rows, data has 24516
Error adding JPM : replacement has 11442 rows, data has 24516
Error adding C : replacement has 12251 rows, data has 24516
Error adding XOM : replacement has 16007 rows, data has 24516
Error adding MCD : replacement has 11442 rows, data has 24516
Error adding GE : replacement has 16007 rows, data has 24516
Error adding NVDA : replacement has 6677 rows, data has 24516
```

The simple approach often fails because stocks have different trading schedules due to holidays, suspensions or listing periods.

2.11.3.2 Approach 2: Robust alignment with match()

```r
# Create data frames with reference dates
Prices = data.frame(date = reference_dates)
Returns = data.frame(date = reference_dates)

# Use match() to handle date misalignment
for(ticker in tickers) {
  ticker_data = data[[ticker]]

  # match() finds positions of reference_dates in ticker_data$date
  # Returns NA when a date is missing from ticker_data
  Prices[[ticker]] = ticker_data$adjusted_close[match(reference_dates, ticker_data$date)]
  Returns[[ticker]] = ticker_data$returns[match(reference_dates, ticker_data$date)]
}

cat("Successfully created aligned data frames\n")
```

```
Successfully created aligned data frames
```

```r
# Check dimensions. ncol(Prices)-1 because the first column is the date
cat("Prices data frame:", nrow(Prices), "days x", ncol(Prices)-1, "securities\n")
```

```
Prices data frame: 24516 days x 10 securities
```

```r
cat("Returns data frame:", nrow(Returns), "days x", ncol(Returns)-1, "securities\n")
```

```
Returns data frame: 24516 days x 10 securities
```

```r
# Check missing values
cat("\nMissing values in Prices:\n")
```

```

Missing values in Prices:
```

```r
for(ticker in tickers) {
  cat(ticker, ":", sum(is.na(Prices[[ticker]])), "NAs\n")
}
```

```
GSPC : 0 NAs
IXIC : 10774 NAs
AAPL : 13262 NAs
MSFT : 14588 NAs
```

```
JPM : 13074 NAs
C : 12265 NAs
XOM : 8509 NAs
MCD : 13074 NAs
GE : 8509 NAs
NVDA : 17839 NAs
```

```r
cat("\nMissing values in Returns:\n")
```

```

Missing values in Returns:
```

```r
for(ticker in tickers) {
  cat(ticker, ":", sum(is.na(Returns[[ticker]])), "NAs\n")
}
```

```
GSPC : 0 NAs
IXIC : 10774 NAs
AAPL : 13262 NAs
MSFT : 14588 NAs
JPM : 13074 NAs
C : 12265 NAs
XOM : 8509 NAs
MCD : 13074 NAs
GE : 8509 NAs
NVDA : 17839 NAs
```

Start this millennium. Convert the integer 20000101 to a date with `ymd(20000101)` (from package `lubridate`) and only look at dates after that.

```r
MinDate=ymd(20000101)
Prices=Prices[Prices$date>MinDate,]
Returns=Returns[Returns$date>MinDate,]

cat("Prices data frame:", nrow(Prices), "days x", ncol(Prices)-1, "securities\n")
```

```
Prices data frame: 6439 days x 10 securities
```

```r
cat("Returns data frame:", nrow(Returns), "days x", ncol(Returns)-1, "securities\n")
```

```
Returns data frame: 6439 days x 10 securities
```

```r
# Check missing values
cat("\nMissing values in Prices:\n")
```

```

Missing values in Prices:
```

```r
for(ticker in tickers) {
  cat(ticker, ":", sum(is.na(Prices[[ticker]])), "NAs\n")
}
```

```
GSPC : 0 NAs
IXIC : 0 NAs
AAPL : 0 NAs
```

```
MSFT : 0 NAs
JPM : 0 NAs
C : 0 NAs
XOM : 0 NAs
MCD : 0 NAs
GE : 0 NAs
NVDA : 0 NAs
```

```r
cat("\nMissing values in Returns:\n")
```

```

Missing values in Returns:
```

```r
for(ticker in tickers) {
  cat(ticker, ":", sum(is.na(Returns[[ticker]])), "NAs\n")
}
```

```
GSPC : 0 NAs
IXIC : 0 NAs
AAPL : 0 NAs
MSFT : 0 NAs
JPM : 0 NAs
C : 0 NAs
XOM : 0 NAs
MCD : 0 NAs
GE : 0 NAs
NVDA : 0 NAs
```

This approach preserves all dates and properly handles missing data, which is important for:

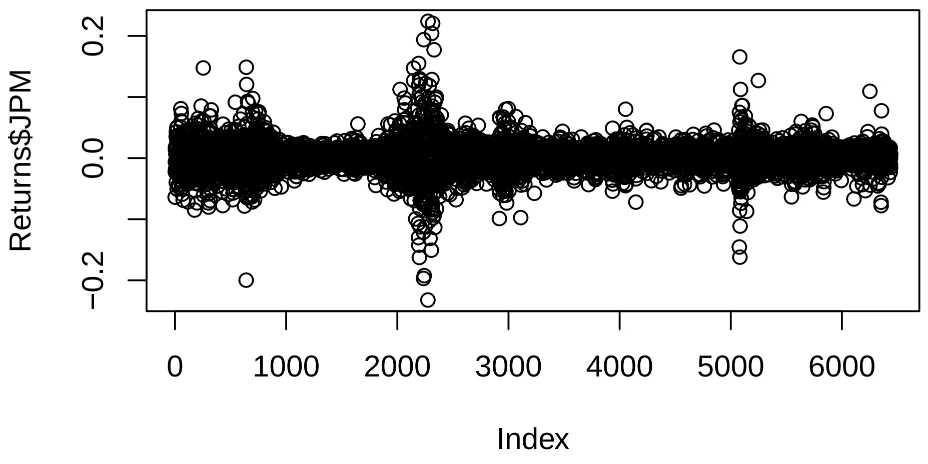
- Accurate correlation calculations (using available data)
- Portfolio analysis across different time periods
- Understanding data availability patterns

2.12 Visualizing financial data

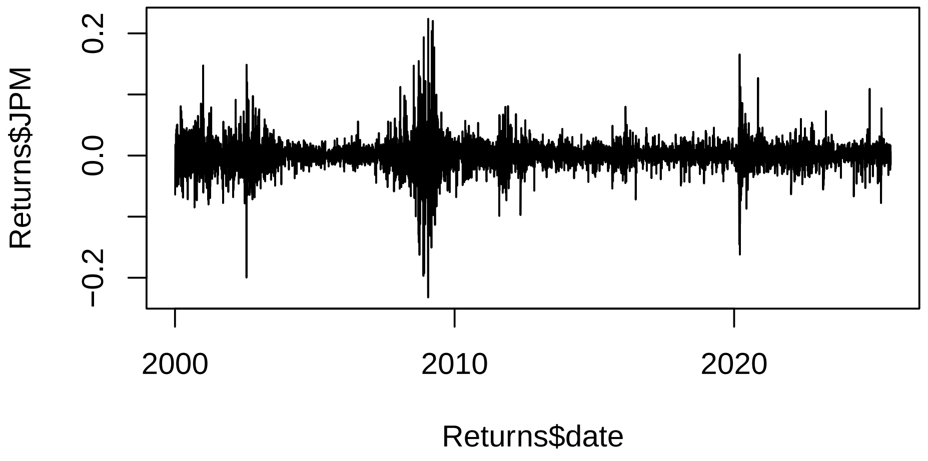
Now create various visualisations of the organised data. Proper visualisation helps with understanding financial data patterns, correlations and risk characteristics. See Plots in the R Notebook for reference.

2.12.1 One asset

```r
# Simple plot of returns
plot(Returns$JPM)
```

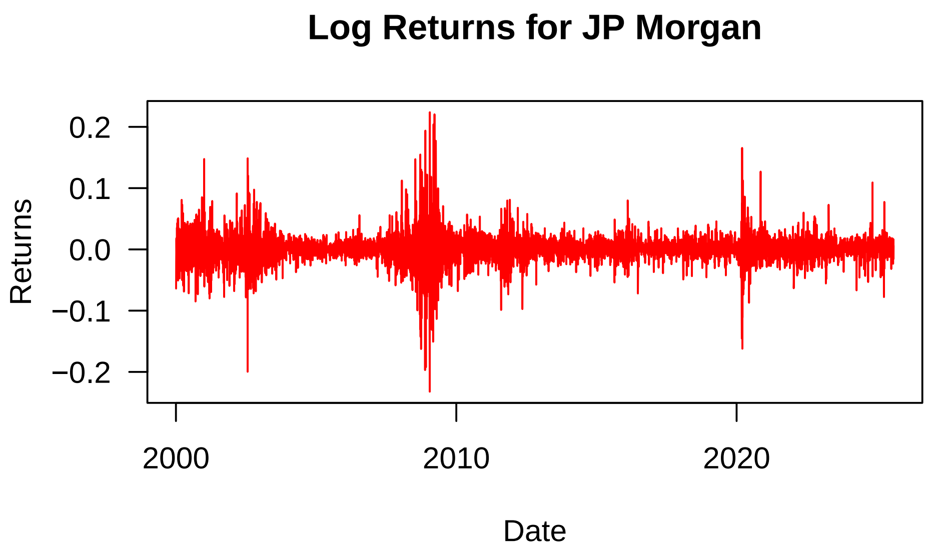


```r
# Line plot with dates
plot(Returns$date, Returns$JPM, type = "l")
```



We can improve this visualisation.

```r
plot(Returns$date, Returns$JPM,
    type = "l",
    main = "Log Returns for JP Morgan",
    ylab = "Returns",
    xlab = "Date",
    col = "red",
    las = 1
)
```

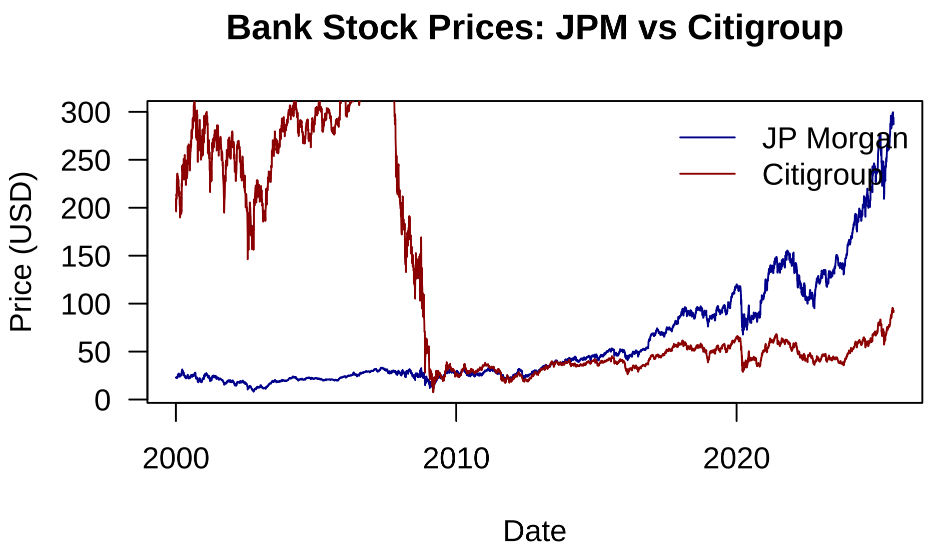


2.12.2 Multiple stocks on one plot

When plotting multiple series, we use `plot()` for the first series and `lines()` to add additional series. However, this approach has its drawbacks.

2.12.2.1 The scaling problem

```r
plot(Prices$date, Prices$JPM,
    type = "l",
    main = "Bank Stock Prices: JPM vs Citigroup",
    ylab = "Price (USD)",
    xlab = "Date",
    col = "darkblue",
    las = 1
)
lines(Prices$date, Prices$C, col = "darkred")
legend("topright",
    legend = c("JP Morgan", "Citigroup"),
    col = c("darkblue", "darkred"),
    lty = 1,
    bty = 'n'
)
```

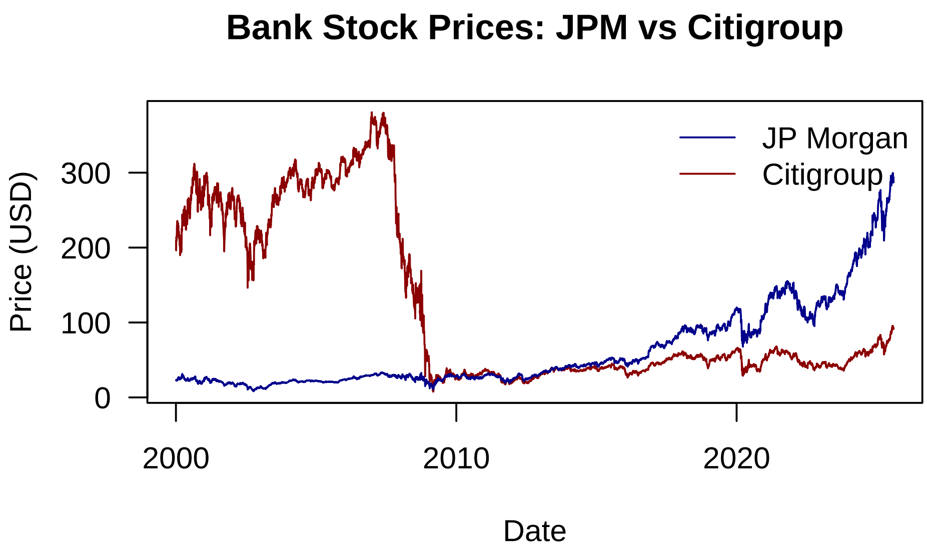


Notice that the Citigroup line is clipped at the top. This happens because the first `plot()` command sets the y-axis range based only on JPM data, ignoring the Citigroup values.

2.12.2.2 Solutions

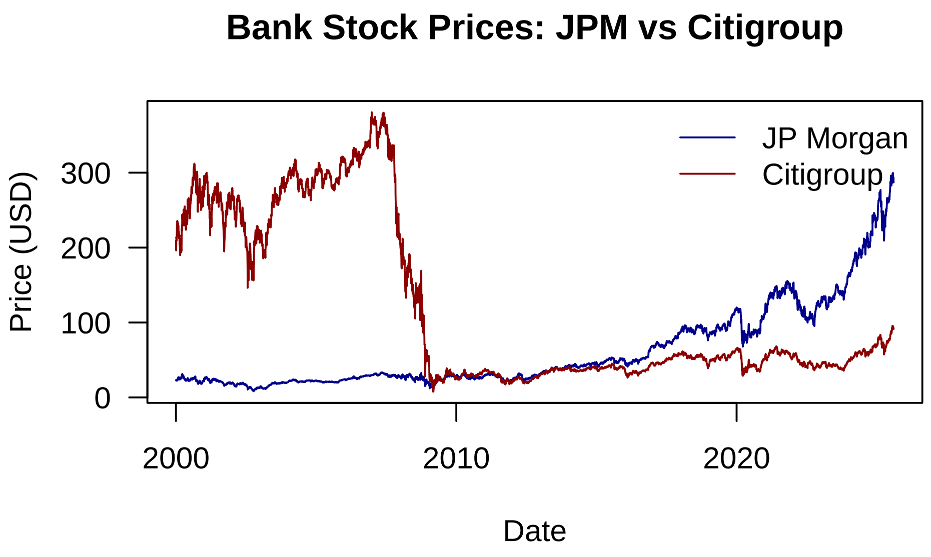
Option 1: Change the plotting order (plot the series with the larger range first)

```r
plot(Prices$date, Prices$C,
    type = "l",
    main = "Bank Stock Prices: JPM vs Citigroup",
    ylab = "Price (USD)",
    xlab = "Date",
    col = "darkred",
    las = 1
)
lines(Prices$date, Prices$JPM, col = "darkblue")
legend("topright",
    legend = c("JP Morgan", "Citigroup"),
    col = c("darkblue", "darkred"),
    lty = 1,
    bty = 'n'
)
```



Option 2: Set explicit y-axis limits using `ylim`

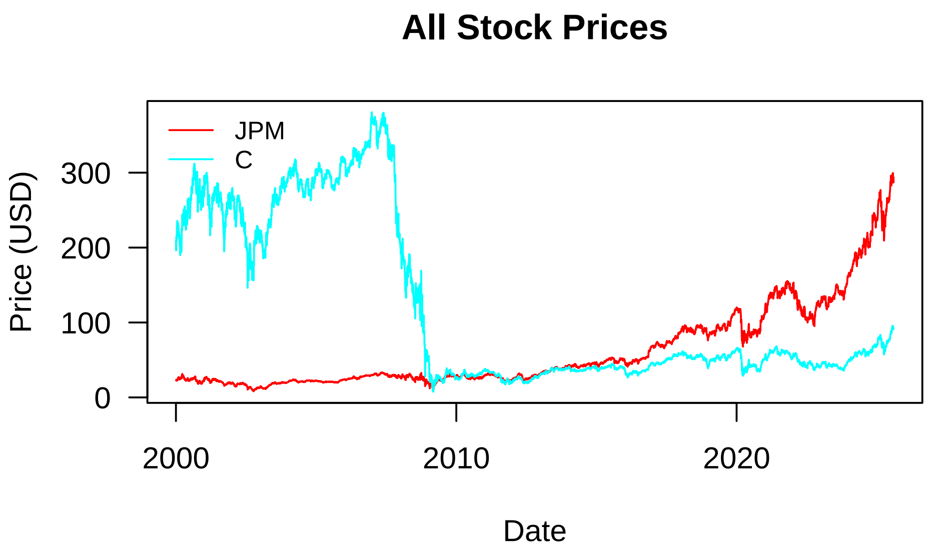
```
plot(Prices$date, Prices$JPM,
    type = "l",
    main = "Bank Stock Prices: JPM vs Citigroup",
    ylab = "Price (USD)",
    xlab = "Date",
    col = "darkblue",
    las = 1,
    ylim = range(c(Prices$JPM, Prices$C), na.rm = TRUE)
)
lines(Prices$date, Prices$C, col = "darkred")
legend("topright",
    legend = c("JP Morgan", "Citigroup"),
    col = c("darkblue", "darkred"),
    lty = 1,
    bty = 'n'
)
```



Best practice: Use `range()` to automatically calculate appropriate limits, as shown above. This works for any number of series and handles missing values properly.

2.12.3 Using matplot for multiple series

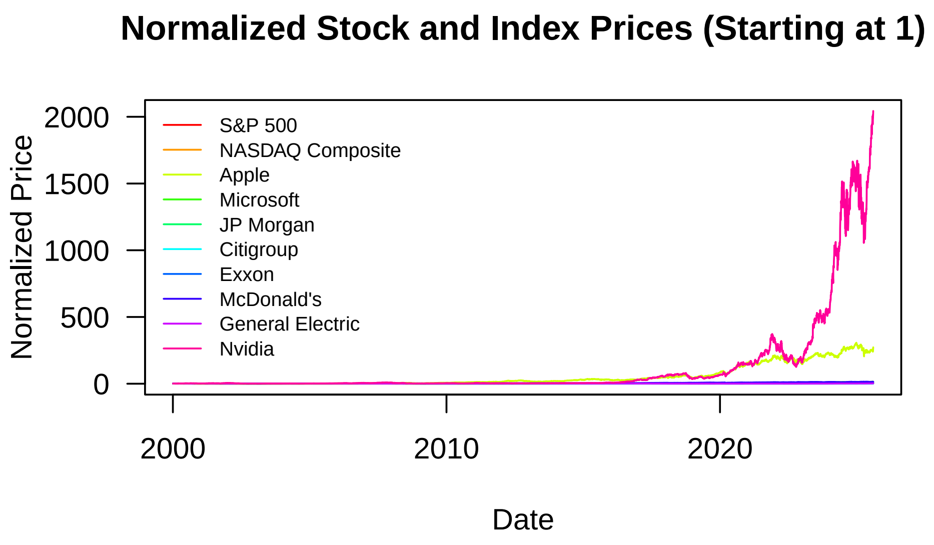
```r
matplot(Prices$date, Prices[, c("JPM", "C")],
    type = "l",
    lty = 1,
    main = "All Stock Prices",
    ylab = "Price (USD)",
    xlab = "Date",
    col = rainbow(2),
    las = 1
)
legend("topleft",
    legend = c("JPM", "C"),
    col = rainbow(2),
    lty = 1,
    bty = 'n',
    cex = 0.8
)
```



2.12.4 Normalised price comparison

Normalising prices shows relative performance regardless of absolute price levels. This is important for portfolio analysis because it reveals which investments performed better percentage-wise. For example, a stock that goes from $10 to $20 has the same relative performance as one that goes from $100 to $200, even though the absolute price changes differ.

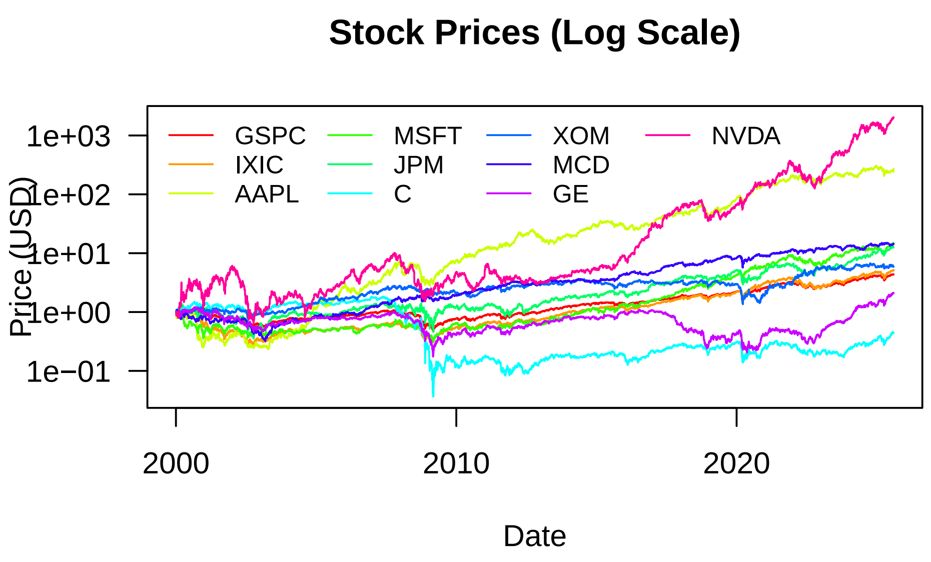
```
# Normalise all prices to start at 1 for comparison
NormalisedPrices = Prices
for(ticker in tickers) {
  first_price = Prices[[ticker]][1]
  NormalisedPrices[[ticker]] = Prices[[ticker]] / first_price
}

# Plot normalised prices
plot_cols = which(names(NormalisedPrices) %in% tickers)
matplot(NormalisedPrices$date, NormalisedPrices[, plot_cols],
    type = "l",
    lty = 1,
    main = "Normalized Stock and Index Prices (Starting at 1)",
    ylab = "Normalized Price",
    xlab = "Date",
    col = rainbow(length(tickers)),
    las = 1
)
legend("topleft",
    legend = security_names[tickers],
    col = rainbow(length(tickers)),
    lty = 1,
    bty = 'n',
    cex = 0.7
)
```



2.12.5 Log scale plot

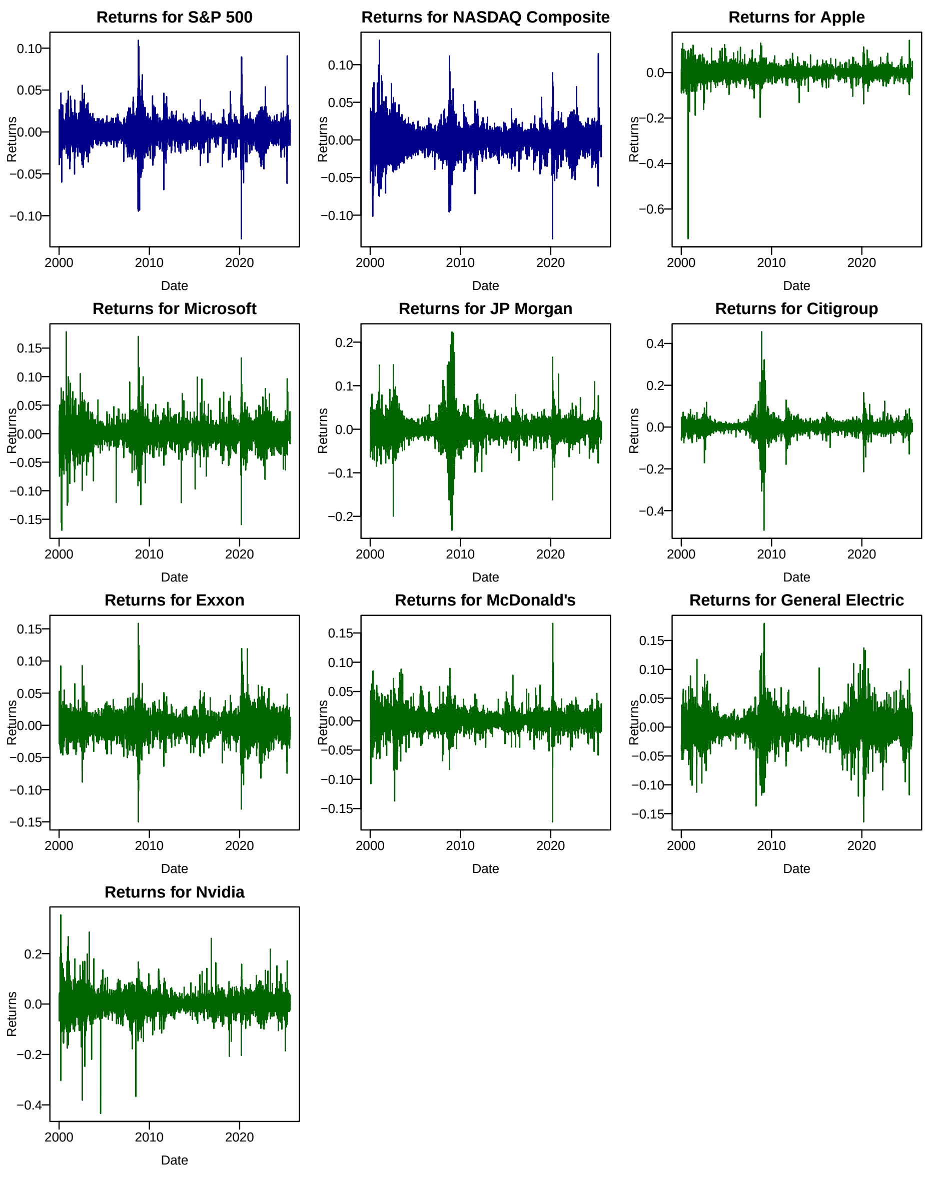
Log scale plotting makes percentage changes appear as equal distances on the chart, which is how financial analysts think about returns. A 10% increase looks the same whether it is from \$10 to \$11 or from \$100 to \$110. This is particularly useful for long-term price analysis when values vary by orders of magnitude.

```
# Plot prices on a log scale
matplot(NormalisedPrices$date, NormalisedPrices[, -1],
    type = "l",
    lty = 1,
    main = "Stock Prices (Log Scale)",
    ylab = "Price (USD)",
    xlab = "Date",
    col = rainbow(length(tickers)),
    las = 1,
    log = "y"
)
legend("topleft",
    legend = tickers,
    col = rainbow(length(tickers)),
    lty = 1,
    bty = 'n',
    cex = 0.8,
    ncol=4
)
```



2.12.6 Subplots for returns

```
# Create a grid of subplots for returns
par(mfrow = c(4, 3), mar = c(3, 3, 2, 1), mgp = c(2, 0.5, 0))
for (ticker in tickers) {
    if (ticker %in% names(Returns)) {
        plot(Returns$date, Returns[[ticker]],
            type = "l",
            ylab = "Returns",
            xlab = "Date",
            main = paste("Returns for", security_names[ticker]),
            col = ifelse(ticker %in% c("GSPC", "IXIC"), "darkblue", "darkgreen"),
            las = 1
        )
    }
}
par(mfrow = c(1, 1))  # Reset to single plot
```



2.13 Exporting plots

For reports and presentations, you can export plots in several ways:

1. Using RStudio: Click on `Export` in the Plot Viewer
2. Programmatically: Use functions like `pdf()`, `png()`, `svg()`

Example:

```
# Save a plot as a PDF
pdf("returns_plot.pdf", width = 10, height = 6)
plot(Returns$date, Returns$AAPL,
    type = "l",
    main = "Apple Returns",
    ylab = "Returns",
    xlab = "Date",
    col = "blue"
)
dev.off()
```

2.14 Saving data frames

After all this data processing, it can be convenient to save the data frames, so we simply load them in subsequent seminars without redoing all the processing. `save()` saves any object as binary data, allowing us to `load` it back unchanged.

```
# Save as RData files
save(Prices, file = "Prices.RData")
save(Returns, file = "Returns.RData")
save(NormalisedPrices, file = "NormalisedPrices.RData")
save(tickers,file = "tickers.RData")
save(security_names,file = "security_names.RData")
```

To load the data in future sessions:

```
# Example of loading saved data
load("Returns.RData")
```

2.14.1 Saving as CSV

We can also save the data frames as CSV files, perhaps to look at in Excel.

```
write.csv(Prices, file = "Prices.csv", row.names = FALSE)
write.csv(Returns, file = "Returns.csv", row.names = FALSE)
```

We can also save them as Excel files.

2.15 Recap

2.15.1 In this seminar, we have covered:

- Finding stock symbols on EOD Historical Data website
- Downloading data individual stocks and examining their structure
- Creating and using functions in R: — Understanding function syntax and arguments — Default parameter values — Return statements — Creating a reusable function `add_returns()` to calculate returns and update data frame
- Efficiently downloading data for multiple stocks including indices
- Creating price and return data frames with proper date alignment
- Handling missing data with NA values
- Saving data as both .RData and .csv files
- Creating various types of plots: — Single time series plots — Multiple series on one plot — Subplot grids — Normalised comparisons — Log scale visualisations
- Customising plots with colours, labels and legends

Some new functions used:

- `add_returns()` — our custom function to add returns
- `merge()` — combine data frames with date alignment
- `unique()` — get unique values from a vector
- `sort()` — sort values in ascending order
- `sum(is.na())` — count missing values
- `matplot()` — plot multiple columns
- `par(mfrow)` — create subplot grids
- `rainbow()` — generate colors
- `lines()` — add lines to existing plots
- `legend()` — add legends to plots

2.16 Optional exercises

1. Practice with the add_returns() function: — Modify the function to calculate simple returns instead of log returns — Create a version that keeps the first row with NA instead of removing it — Add an argument to choose between simple and log returns

2. Data exploration: — Download additional stocks (e.g., AMZN, GOOGL, META) — Check how many missing values each has — Which stocks have the most complete data?

3. Return analysis: — Calculate the average return for each stock using the Returns data frame — Find which stock has the highest volatility (standard deviation) — On which dates do all stocks have data available?

4. Visualization practice: — Create a subplot showing prices for tech stocks vs. financial stocks — Make a normalised price plot starting from a specific date (e.g., 2020-01-01) — Use different line styles (lty = 1, 2, 3) to distinguish between sectors

5. Missing data investigation: — Plot the number of stocks with available data for each date — Identify periods where multiple stocks have missing data — Create a heatmap showing data availability (1 for available, 0 for NA)

6. Index comparison: — Plot individual stock returns against the S&P 500 returns — Which stocks outperformed the index? — Calculate the correlation between each stock and the S&P 500

7. Event analysis - COVID and Trump tariffs:

    - Extract data for specific periods:
        - Trump tariffs: January 2025 to now
        - COVID crash: February 2020 to December 2020
    - Calculate cumulative returns for each stock during these periods
    - Which sectors were most affected by each event?
    - Create before/after plots showing price changes around March 2020
    - Compare volatility (standard deviation of returns) before and during these events