gwp1_grp_8781_Assignment_Notebook

April 15, 2025

Group Members

1. Halkano Molu Guracha
2. Biswajit Paljit
3. Joel Muchemeranwa

0.0.1 Installing Dependancies

[1]:
```
!pip install nelson_siegel_svensson
!pip install yfinance
```

```
Collecting nelson_siegel_svensson
  Downloading nelson_siegel_svensson-0.5.0-py2.py3-none-any.whl.metadata (6.7
kB)
Requirement already satisfied: Click>=8.0 in /usr/local/lib/python3.11/dist-
packages (from nelson_siegel_svensson) (8.1.8)
Requirement already satisfied: numpy>=1.22 in /usr/local/lib/python3.11/dist-
packages (from nelson_siegel_svensson) (2.0.2)
Requirement already satisfied: scipy>=1.7 in /usr/local/lib/python3.11/dist-
packages (from nelson_siegel_svensson) (1.14.1)
Requirement already satisfied: matplotlib>=3.5 in
/usr/local/lib/python3.11/dist-packages (from nelson_siegel_svensson) (3.10.0)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.11/dist-packages (from
matplotlib>=3.5->nelson_siegel_svensson) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-
packages (from matplotlib>=3.5->nelson_siegel_svensson) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.11/dist-packages (from
matplotlib>=3.5->nelson_siegel_svensson) (4.57.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
/usr/local/lib/python3.11/dist-packages (from
matplotlib>=3.5->nelson_siegel_svensson) (1.4.8)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.11/dist-packages (from
matplotlib>=3.5->nelson_siegel_svensson) (24.2)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-
packages (from matplotlib>=3.5->nelson_siegel_svensson) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.11/dist-packages (from
matplotlib>=3.5->nelson_siegel_svensson) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.11/dist-packages (from
matplotlib>=3.5->nelson_siegel_svensson) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-
```

```
packages (from python-dateutil>=2.7->matplotlib>=3.5->nelson_siegel_svensson)
(1.17.0)
Downloading nelson_siegel_svensson-0.5.0-py2.py3-none-any.whl (9.9 kB)
Installing collected packages: nelson_siegel_svensson
Successfully installed nelson_siegel_svensson-0.5.0
Requirement already satisfied: yfinance in /usr/local/lib/python3.11/dist-
packages (0.2.55)
Requirement already satisfied: pandas>=1.3.0 in /usr/local/lib/python3.11/dist-
packages (from yfinance) (2.2.2)
Requirement already satisfied: numpy>=1.16.5 in /usr/local/lib/python3.11/dist-
packages (from yfinance) (2.0.2)
Requirement already satisfied: requests>=2.31 in /usr/local/lib/python3.11/dist-
packages (from yfinance) (2.32.3)
Requirement already satisfied: multitasking>=0.0.7 in
/usr/local/lib/python3.11/dist-packages (from yfinance) (0.0.11)
Requirement already satisfied: platformdirs>=2.0.0 in
/usr/local/lib/python3.11/dist-packages (from yfinance) (4.3.7)
Requirement already satisfied: pytz>=2022.5 in /usr/local/lib/python3.11/dist-
packages (from yfinance) (2025.2)
Requirement already satisfied: frozendict>=2.3.4 in
/usr/local/lib/python3.11/dist-packages (from yfinance) (2.4.6)
Requirement already satisfied: peewee>=3.16.2 in /usr/local/lib/python3.11/dist-
packages (from yfinance) (3.17.9)
Requirement already satisfied: beautifulsoup4>=4.11.1 in
/usr/local/lib/python3.11/dist-packages (from yfinance) (4.13.3)
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.11/dist-
packages (from beautifulsoup4>=4.11.1->yfinance) (2.6)
Requirement already satisfied: typing-extensions>=4.0.0 in
/usr/local/lib/python3.11/dist-packages (from beautifulsoup4>=4.11.1->yfinance)
(4.13.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.11/dist-packages (from pandas>=1.3.0->yfinance) (2.8.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-
packages (from pandas>=1.3.0->yfinance) (2025.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinance) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
packages (from requests>=2.31->yfinance) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinance) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests>=2.31->yfinance)
(2025.1.31)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-
packages (from python-dateutil>=2.8.2->pandas>=1.3.0->yfinance) (1.17.0)
```

[2]:
```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
from nelson_siegel_svensson.calibrate import calibrate_ns_ols
import numpy as np
from numpy import linalg as LA
import seaborn as sns
```

1 Task 2

1.0.1 Q2. a. and Q2. b.

We selected the country - India. The following dataset contains the daily yields of the Indian bonds with the following maturities from 2006 to 2025(current).

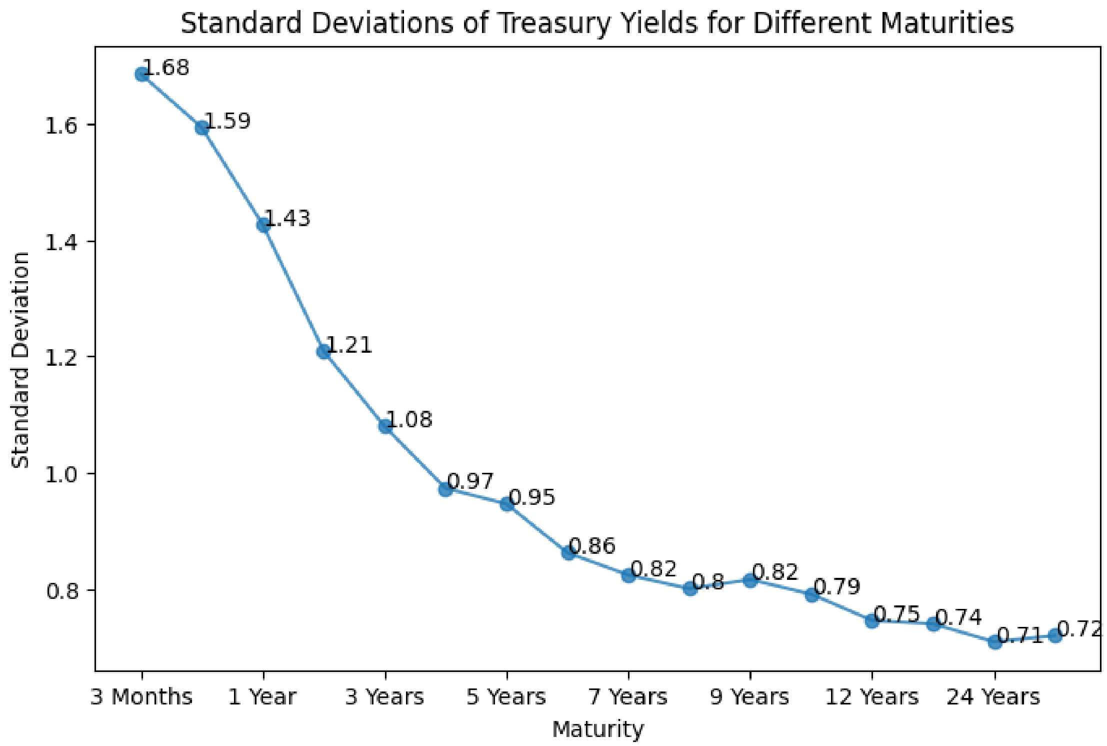
Maturity	Source
3 Months	India 3-Month Bond Yield
6 Months	India 6-Month Bond Yield
1 Year	India 1-Year Bond Yield
2 Year	India 2-Year Bond Yield
3 Year	India 3-Year Bond Yield
4 Year	India 4-Year Bond Yield
5 Year	India 5-Year Bond Yield
6 Year	India 6-Year Bond Yield
7 Year	India 7-Year Bond Yield
8 Year	India 8-Year Bond Yield
9 Year	India 9-Year Bond Yield
10 Year	India 10-Year Bond Yield
12 Year	India 12-Year Bond Yield
15 Year	India 15-Year Bond Yield
24 Year	India 24-Year Bond Yield
30 Year	India 30-Year Bond Yield

[5]:
```
df = pd.read_csv('C:/Users/Gast01/WQU-GWP1/India Bond Yield Data.csv') # Change
 ↪the file path as the path in your device.

df.index = pd.to_datetime(df['Date'])

df = df.drop('Date', axis=1)
df = df.dropna()
```

[6]:
```
def plot_std(df):
    """
    Plotting the standard deviation of the treasury yields for different
 ↪maturities
    """
```

```
    y_std = df.std()
    fig, ax = plt.subplots()
    y_std.plot(figsize = (8,5),marker='o', title='Standard Deviations of Treasury␣
↪Yields for Different Maturities', alpha=0.8)
    plt.xlabel("Maturity")
    plt.ylabel("Standard Deviation")
    for i in range(len(y_std)):
        ax.annotate(str(round(y_std.iloc[i],2)),xy=(i,y_std.iloc[i]))
    plt.show()
```

[7]:
```
plot_std(df)
```



[8]:
```
def plot_yield_curve(date, fig_n):
    """
    Plotting the yield curve for a given date
    """

    maturities = df.columns # Maturities
    fig, ax = plt.subplots(figsize=(6.15, 4))
    ax.plot(maturities, df.loc[date], marker='D', label='Yield Curve at ' +␣
↪date)
```

```
    ax.set_yticklabels(['{:.2f}%'.format(y) for y in ax.get_yticks()])
    ax.set_xticks(range(len(maturities)))
    labels = [m if i % 5 == 0 else '' for i, m in enumerate(maturities)]
    ax.set_xticklabels(labels)

    # Add labels and title
    ax.set_xlabel('Maturity')
    ax.set_ylabel('Yield')
    ax.set_title(fig_n+f'Treasury Yield Curve as of {date}')

    # Show the plot
    plt.grid(False)
    plt.show()
```

1.0.2 Q2. c.

Fitting a Nelson Siegel Model on all the maturities on a specific date and comparing it with the original yield curve as of that date

[9]:
```
def fit_ns(t, y, tau0=1.0):
    """
    Fitting the Nelson-Siegel model to the yield curve data for a given date.
    """

    curve, status = calibrate_ns_ols(t, y, tau0=1.0)  # starting value of 1.0 for␣
↪the optimization of tau
    assert status.success

    print(curve)

    return curve
```

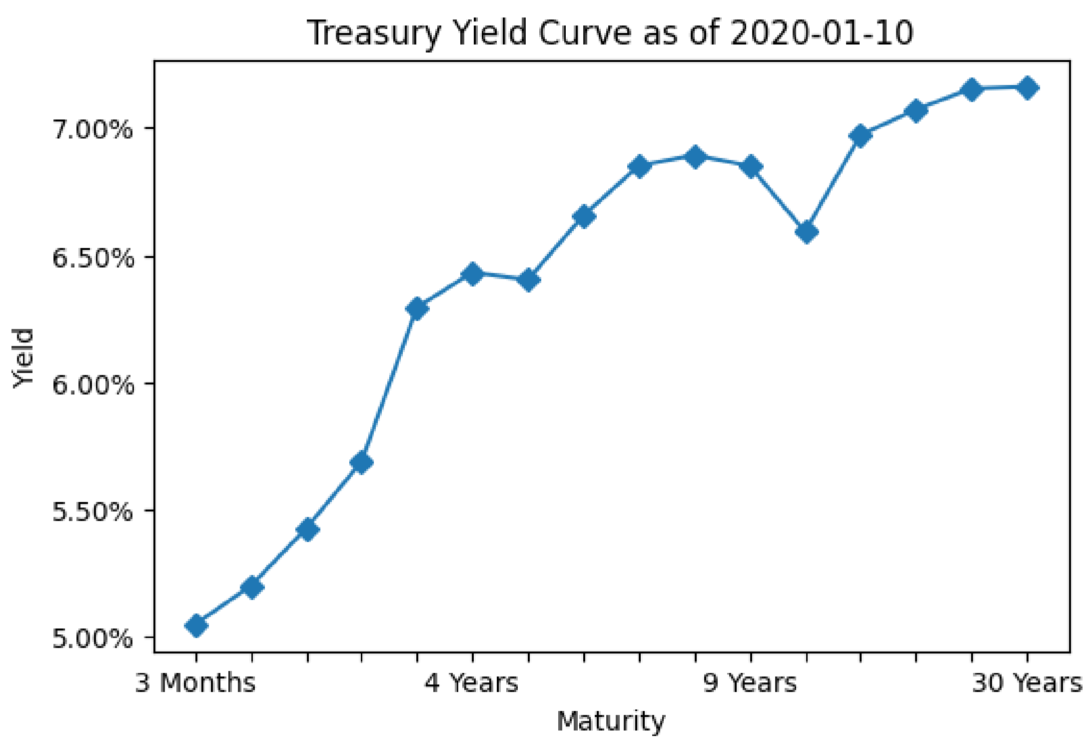
[10]:
```
def plot_ns(date, y_hat, t_hat):
    """
    Plotting the yield curve for a given date.
    """

    plt.plot(t_hat, y_hat(t_hat))
    plt.xlabel("Maturity")
    plt.ylabel("Yield")
    plt.title(f"NS Model Result as of {date}")

    plt.show()
```

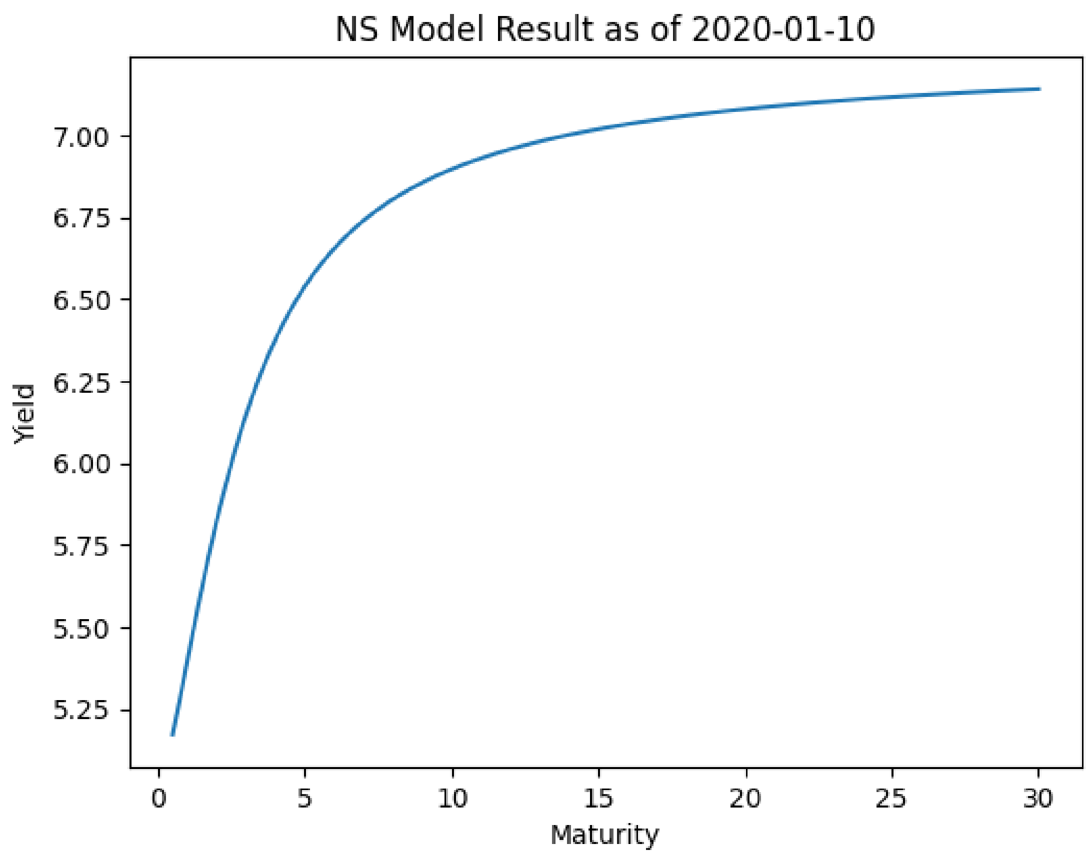
[11]:
```
date = "2020-01-10"
t = np.array([0.25,0.5,1,2,3,4,5,6,7,8,9,10,12,15,24,30])
```

```
y = np.array(df.loc[date])
curve = fit_ns(t, y, tau0=1.0)
y_hat = curve
t_hat = np.linspace(0.5,30,100)
plot_yield_curve(date,'')
plot_ns(date, y_hat, t_hat)
```

```
NelsonSiegelCurve(beta0=np.float64(7.262536953580489),
beta1=np.float64(-2.2736710165571905), beta2=np.float64(-1.7792170270160836),
tau=np.float64(0.9062504122308159))
```

```
<ipython-input-8-f6116200d263>:10: UserWarning: set_ticklabels() should only be
used with a fixed number of ticks, i.e. after set_ticks() or using a
FixedLocator.
  ax.set_yticklabels(['{:.2f}%'.format(y) for y in ax.get_yticks()])
```





1.0.3 Q2. f.

In the solution above, we have fit a Nelson Siegel Model on the yield curve as of 2020-01-10. The result `NelsonSiegelCurve(beta0=np.float64(7.262536960509667), beta1=np.float64(-2.273671047246356), beta2=np.float64(-1.779216868209846), tau=np.float64(0.9062504579008243))` indicates that the β_0 = 7.26 showing the level of the yield curve. β_1 = -2.27 shows the slope of the yield curve and the $\beta 2$ = -1.77 shows the shape or the curvature of the yield curve. the decay rate indicated by τ = 0.91 showing a slow rate of decay.

A level of 7.26 shows that the long term expectation of the yields to be around 7%.

A negative slope can be attributed to the fact that there are dips in the long term yields, for example the 10 year yield is smaller than the 6 years, 7years and 8 years yields.

A negative curvature indicates a concave behaviour. The yields started rising steep in the short run, but flatten over a longer period.

This interpretation is in line with the mid pandemic situation in 2020. Complete economic shutdowns has raised riskiness of the short term borrowings, hence the rise in yields sharply.

A smaller value of τ (close to 1) indicates that the effects of the slope and curvature parameters decay relatively slowly as maturity increases. This suggests that the short-to-medium-term rates have a strong influence on the shape of the curve, but as maturity increases, the curve flattens, and the level dominates.

1.0.4 Q2. d. and Q2. f.

Fitting a Cubic Spline Model on all the maturities on a specific date and comparing it with the original yield curve as of that date.

Since there are 16 maturities, there would be 15 splines that pass through the 16 points. The splines can be shown as follows:

$$f(x) = a_1x^3 + b_1x^2 + c_1x + d_1, \quad \text{when } 0.25 \leq x \leq 0.5$$

$$f(x) = a_2x^3 + b_2x^2 + c_2x + d_2, \quad \text{when } 0.5 \leq x \leq 1$$

$$f(x) = a_3x^3 + b_3x^2 + c_3x + d_3, \quad \text{when } 1 \leq x \leq 2$$

and so on…

From the above equations, we have (15 × 4 = 60) unknowns. Hence, we need 60 equations to solve for the parameters.

Thus plugging each boundary, we get 30 equations as shown below:

$$\begin{gathered}
a_1(0.25)^3 + b_1(0.25)^2 + c_1(0.25) + d_1 = 5.05 \quad (1) \\
a_1(0.5)^3 + b_1(0.5)^2 + c_1(0.5) + d_1 = 5.2 \quad (2) \\
a_2(0.5)^3 + b_2(0.5)^2 + c_2(0.5) + d_2 = 5.2 \quad (3) \\
a_2(1)^3 + b_2(1)^2 + c_2(1) + d_2 = 5.42 \quad (4) \\
a_3(1)^3 + b_3(1)^2 + c_3(1) + d_3 = 5.42 \quad (5) \\
a_3(2)^3 + b_3(2)^2 + c_3(2) + d_3 = 5.69 \quad (6)
\end{gathered}$$

Now since each interirior point is a part of 2 splines, their slopes and curvatures must be the same. Therefore, their first order derivatives and their second order derivatives should be the same. Since there are (16 - 2 = 14) interior points, we get 14 first order equations, and 14 second order equations.

The first order equations are as follows:

$$\begin{gathered}
3a_1(0.5)^2 + 2b_1(0.5) + c_1 = 3a_2(0.5)^2 + 2b_2(0.5) + c_2 \quad (31) \\
3a_2(1)^2 + 2b_2(1) + c_2 = 3a_3(1)^2 + 2b_3(1) + c_3 \quad (32)
\end{gathered}$$

The second order equations are as follows:

$$6a_1(0.5) + 2b_1 = 6a_2(0.5) + 2b_2 \quad (45)$$

$$6a_2(1) + 2b_2 = 6a_3(1) + 2b_3 \quad (46)$$

In total now we have 58 equations. Finally we assume the "Natural End Condition" and consider the secodn order derivatives of the exterior p[oints to be 0/ Since there are 2 exterior points, we get 2 equations, which gives us full 60 equations.

$$6a_1(0.25) + 2b_1 = 0 \quad (59)$$
$$6a_3(30) + 2b_3 = 0 \quad (60)$$

To calculate the cubic spline model, we use the `CubicSpline` function from the **SciPy** library, typically imported as `cs`.

The command is:

`cs = CubicSpline(t, y, bc_type=((2, 0.0), (2, 0.0)))`

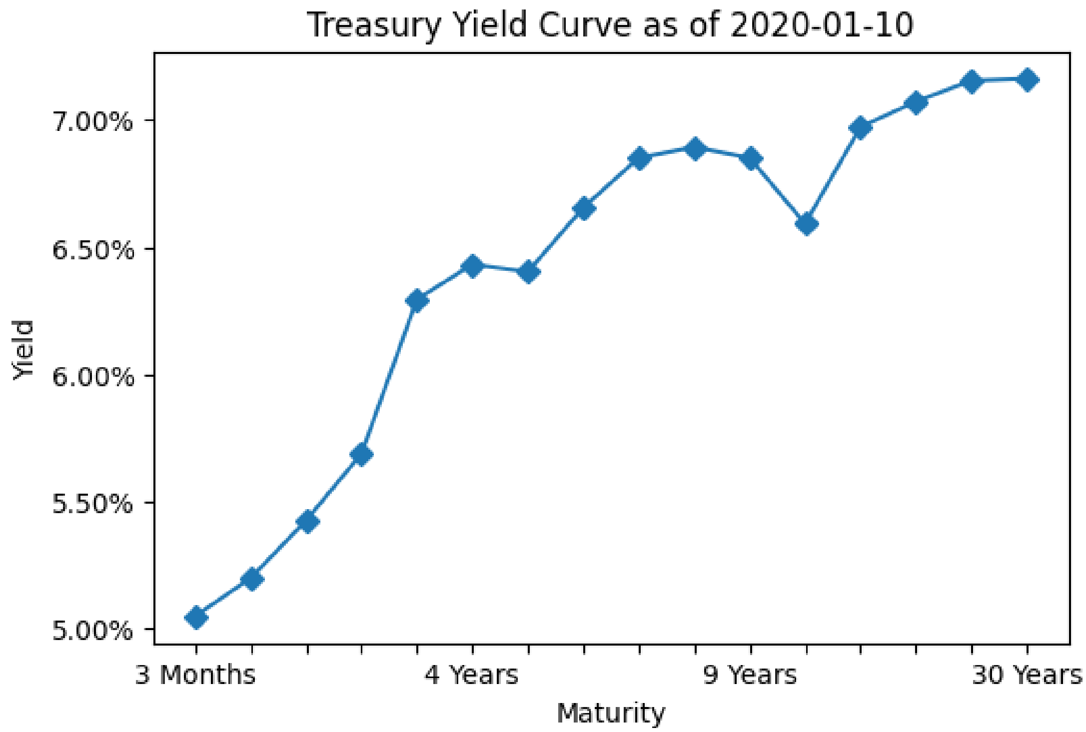
where:

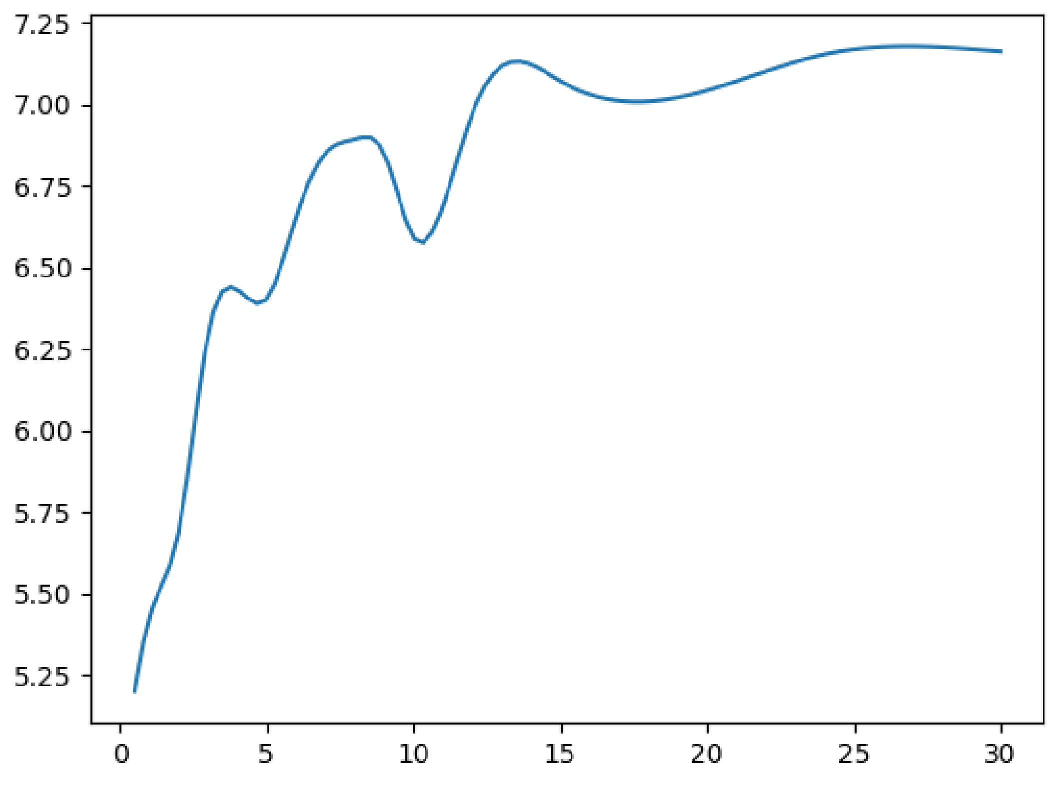
- `t` = maturities
- `y` = yields as of a specific date
- `bc_type=((2, 0.0), (2, 0.0))` sets the **natural end condition** by specifying that the second derivative at the two endpoints is zero.

[12]:
```python
def fit_cs(t, y, t_hat):
  """
  Fitting a Cubic Spline Model on all the maturities on a specific date.
  """
  cs = CubicSpline(t, y, bc_type=((2, 0.0), (2, 0.0)))
  interpolated_yields = cs(t_hat)
  plt.plot(t_hat, interpolated_yields)
```

[13]:
```python
date = "2020-01-10"
t = np.array([0.25,0.5,1,2,3,4,5,6,7,8,9,10,12,15,24,30])
y = np.array(df.loc[date])
t_hat = np.linspace(0.5,30,100)
plot_yield_curve(date,'')
fit_cs(t, y, t_hat)
```

```
<ipython-input-8-f6116200d263>:10: UserWarning: set_ticklabels() should only be
used with a fixed number of ticks, i.e. after set_ticks() or using a
FixedLocator.
  ax.set_yticklabels(['{:.2f}%'.format(y) for y in ax.get_yticks()])
```


Treasury Yield Curve as of 2020-01-10
7.00%
6.50%
6.00%
5.50%
5.00%
Yield
3 Months
4 Years
9 Years
30 Years
Maturity


7.25
7.00
6.75
6.50
6.25
6.00
5.75
5.50
5.25
0
5
10
15
20
25
30

1.0.5 Q2. e.

Fit: - NS Model is a parametric model with defined parameters β_0, β_1 and β_2. The model smoothened the trend of the yield curve and just fit the general level witht the slope and curvature of the curve. It failed to capture the nuances of the mid term maturities. Features like a lower yield in the 10 year maturity did not get captured in the NS Model.

- Cubic Spline is a non parametric model which fits a spline between each point individually and thus captures a much more nuanced view of the yield curve. The fit is indeed much better than the NS Model.

Interpretation: - Since NS is a parametric approach, its parameters have a strict economic interpretation. - β_0 : Level - β_1 : Slope - β_2 : Curvature - τ : Decay Rate

It is helpful in making economic decision based on these parameters.

- Cubic Spline takes a non parametric approach and as a result does not have a direct economic interpretation. Its use cases are when one needs a data driven approach to mapping the yield movements, with a certain degree of smoothing involved. However, It is very succeptible to overfitting. Thus one must be careful towards the model complexity-overfitting tradeoff.

1.0.6 Q2. g.

Indeed Smoothing data can be considered unethical, as discussed in M2 L4. Howver, whether or not the NS model smoothing is unethical, depends on the specific use cases.

NS Model is a parametric approach to smoothen out the yield curve based on its broad economic parameters, $\beta_0, \beta_1, \beta_2$ and τ. It aims to provide a simpified picture of the economic conditions based on the yields, to provide insights on the expected trends in the market both in the short and long term. In this case the smoothing out is not done to hide variability, it is done to cancel out the noise to identify trends. It is not unethical to smoothen the curve via NS Model in this case.

However, say in a situation where the user does the simplification, but fails to disclose it, with a purpose of hiding variability, or hiding a sharp downturn in the yields, it becomes unethical.

2 Task 3

Q3. a & b) Generating 5 Gausian uncorrelated random variables and running Principal Components on Covariance Matrix

[14]:
```
np.random.seed(42)
sim_yld = np.random.normal(loc = 0, scale = 0.01, size = (200,5))
yld = pd.DataFrame(sim_yld, columns = ['3 Months', '1 Year', '3 Years', '10␣
↪Years', '30 Years'])
mean_yld = yld.mean()
std_yld = yld.std()
yc_standardised = (yld - mean_yld) / std_yld
std_data_cov = yc_standardised.cov()
eigenvalues, eigenvectors = LA.eig(std_data_cov)
```

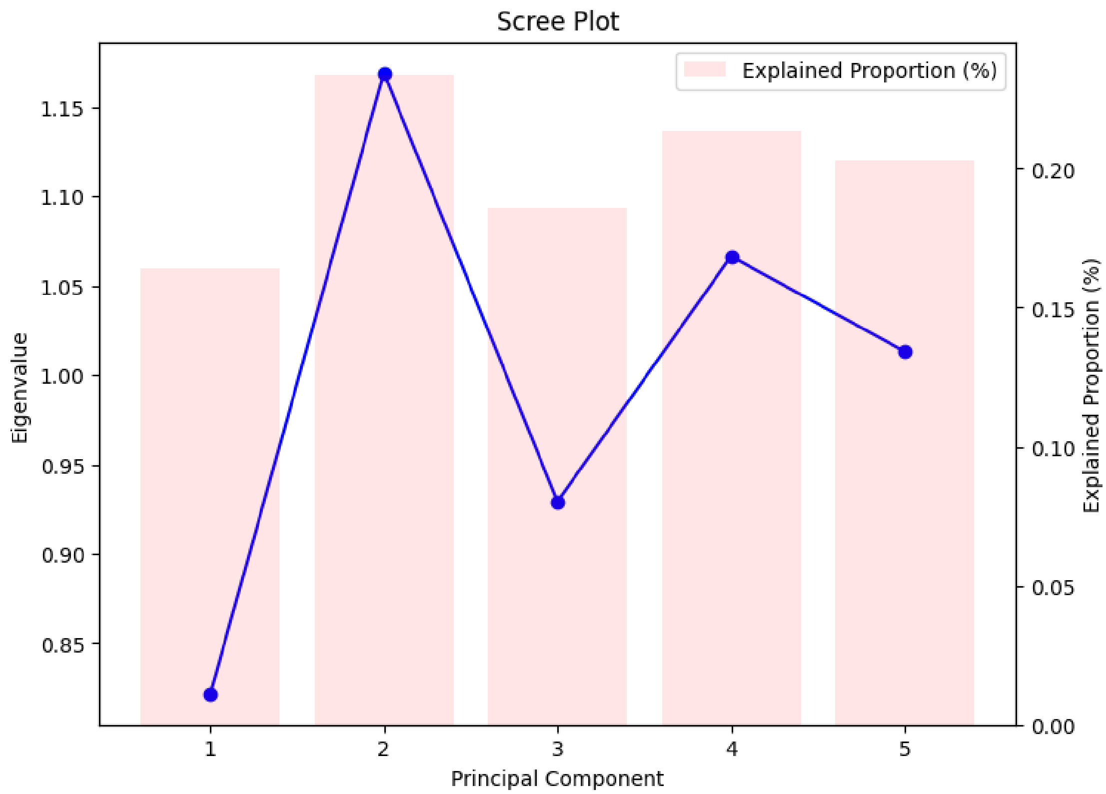
```
principal_components = yc_standardised.dot(eigenvectors)
principal_components.columns = ["PC_1","PC_2","PC_3","PC_4","PC_5"]
df_eigval = pd.DataFrame({"Eigenvalues":eigenvalues}, index=range(1,6))
df_eigval["Explained proportion"] = df_eigval["Eigenvalues"] / np.
 ↪sum(df_eigval["Eigenvalues"])
df_eigval.style.format({"Explained proportion": "{:.2%}"})
```

[14]: <pandas.io.formats.style.Styler at 0x7d09633dc790>

Q3. c) The above df shows the principal componentsw described by their eigen values and the amount of explanation of the variance they respectively do. As we can see that in case of random variables, which are uncorrelated, the 5 principal components more or less describe an equal proportion of the variance. This can be owed to the fact that the multivariate distributions are i.i.d across the observations.

Q3 d) Plotting the Scree plot of the variance explained by each component.

[15]:
```
# Plotting the Scree plot

plt.figure(figsize=(8, 6))
plt.plot(range(1, 6), df_eigval['Eigenvalues'], marker='o', linestyle='-',␣
 ↪color='b', label='Eigenvalues')
plt.title("Scree Plot")
plt.xlabel("Principal Component")
plt.ylabel("Eigenvalue")
plt.xticks(range(1, 6))

# Step 2: Plotting explained proportion as bars

plt.twinx()
plt.bar(range(1, 6), df_eigval['Explained proportion'], alpha=0.1, color='r',␣
 ↪label="Explained Proportion (%)")
plt.ylabel("Explained Proportion (%)")

plt.grid(False)
plt.legend(loc="upper right")
plt.show()
```



Here we can see that the explained variance does not reduce as the number of principal components increase. Infact they explain more or less similar proportions of the variance.

Q3. e & f) Loading actual data of government securities, for the past 6 months and converting yields into yield changes.

[17]:
```
df = pd.read_csv('C:/Users/Gast01/WQU-GWP1/India Bond Yield Data.csv')
df.index = pd.to_datetime(df['Date'])
df_sec = df[df.index >= (df.index.max() - pd.DateOffset(months=6))]

# Selecting 5 securities
df_sec = df_sec[['3 Months', '1 Year', '3 Years', '10 Years', '30 Years']]

# Converting yields into yield changes and dropping NaNs
df_pct_chg = df_sec.pct_change().dropna()
```

Q3. g) Running Principal Components on original yields, using covariance matrix.

[18]:
```
yld_chg_mean = df_pct_chg.mean()
yld_chg_std = df_pct_chg.std()
standardized_yld_chag = (df_pct_chg - yld_chg_mean) / yld_chg_std
std_yld_chg_cov = standardized_yld_chag.cov()
eigenvalues_yld, eigenvectors_yld = LA.eig(std_yld_chg_cov)
```

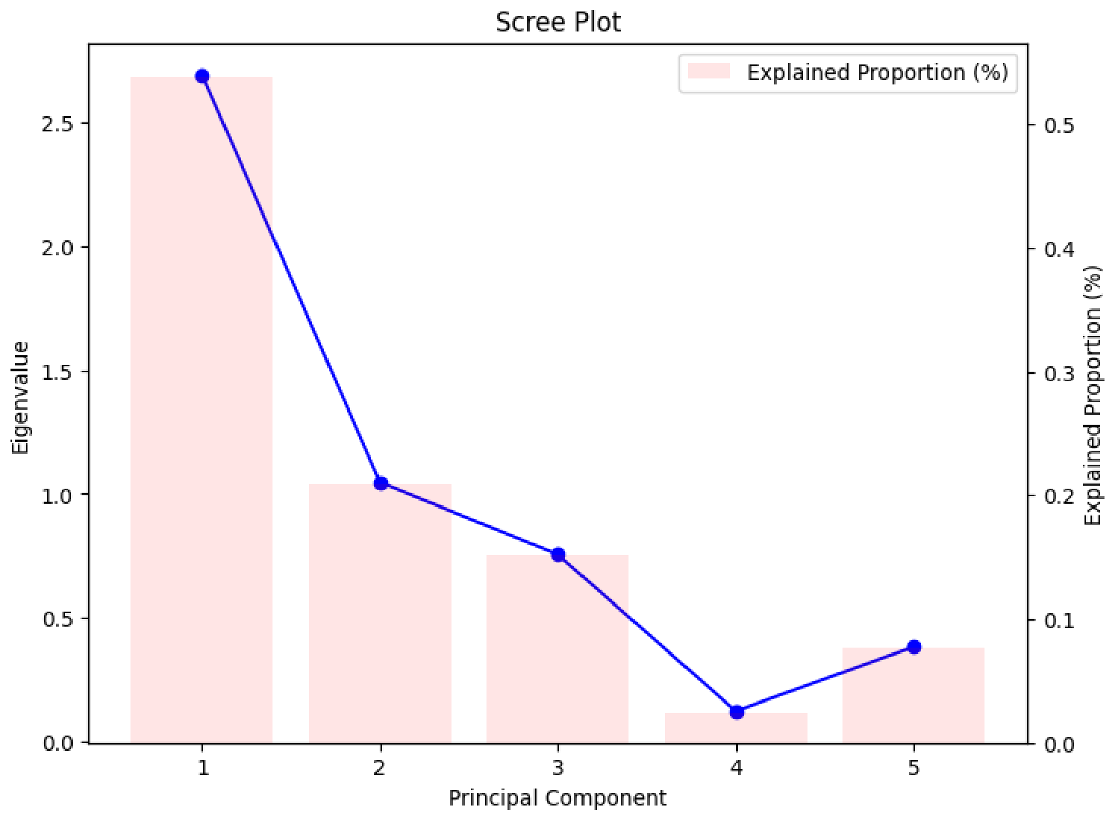
```
principal_components = standardized_yld_chag.dot(eigenvectors_yld)
principal_components.columns = ["PC_1","PC_2","PC_3","PC_4","PC_5"]
df_eigval_yld = pd.DataFrame({"Eigenvalues":eigenvalues_yld}, index=range(1,6))
df_eigval_yld["Explained proportion"] = df_eigval_yld["Eigenvalues"] / np.
↪sum(df_eigval_yld["Eigenvalues"])
df_eigval_yld.style.format({"Explained proportion": "{:.2%}"})
```

[18]: <pandas.io.formats.style.Styler at 0x7d0965c5bf10>

Q3. h) From the above df we can see a different picture regarding the eigenvectors and their explanation of the variance. The first PC explains more than 50% of the variance in the yields. This is the "Level" of the yield curve. This shows the long term expectations of the yields. The secodn PC explains almost 21% of the variance. We can interpret this as the "Slope" of the yield curve. Finally the 3rd PC explains about 15% of the variance, and this can be attributed to the "Curvature" of the yield curve, or how fast the slope is changing. The last 2 PCs explain relatively lower proportion of the variance, since the level, slope and curvature are the 3 main principal components.

Q3. i) Producing Scree Plot

[19]:
```
# Plotting the Scree plot

plt.figure(figsize=(8, 6))
plt.plot(range(1, 6), df_eigval_yld['Eigenvalues'], marker='o', linestyle='-',␣
↪color='b', label='Eigenvalues')
plt.title("Scree Plot")
plt.xlabel("Principal Component")
plt.ylabel("Eigenvalue")
plt.xticks(range(1, 6))
plt.grid(False)

# Plotting explained proportion as bars
plt.twinx()
plt.bar(range(1, 6), df_eigval_yld['Explained proportion'], alpha=0.1,␣
↪color='r', label="Explained Proportion (%)")
plt.ylabel("Explained Proportion (%)")
plt.legend(loc="upper right")

# Show the plot
plt.show()
```



Q3. j.

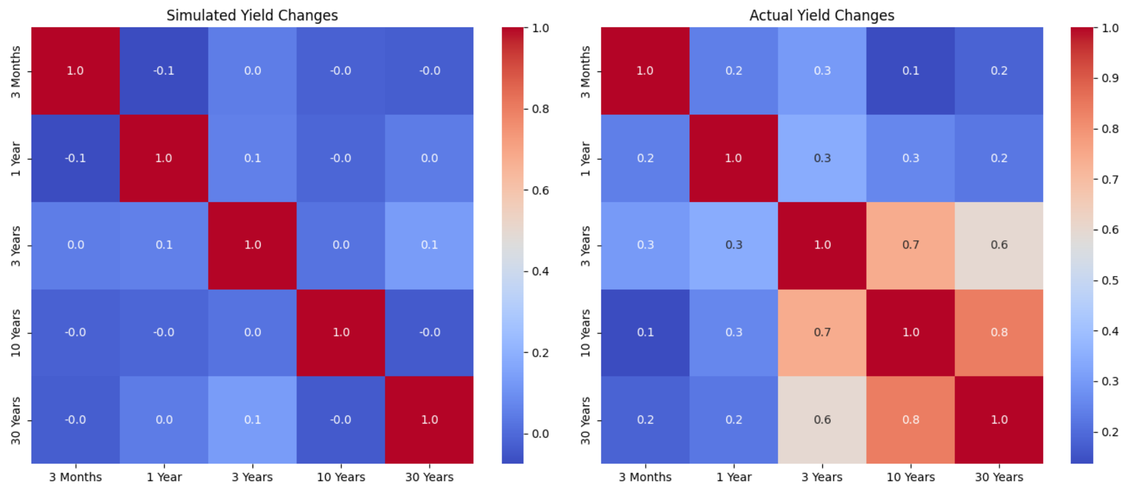
The above scree plot is more in line with our expectations. The plot can be read as an elbow method. The elbow forming when the first principal component is explaining the highest variance and as the number of PCs increase, they explain lesser and lesser of the variance of the yield curve. In our case, the first three principal components explain about 92% of the data.

This observation is different than the uncorrelated gausian random variables case because the data was generated such that the random variables had a same variance. As a result there is no dominant direction, and the variance is equally spread across the 5 eigenvalues, so the PC algorithm finds similar eigenvalues for the 5 PCs.

Since principal components are dimentionality reduction techniques, they work well when the variables are closely correlated. From the following figure we see that the yield changes from the actual data is more correlated than the gaussian data generation process. Hence PCA performed better on the real data.

[20]:
```
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Simulated yield changes
sns.heatmap(yc_standardised.corr(), annot=True, cmap='coolwarm', fmt=".1f",␣
↪ax=axes[0])
axes[0].set_title('Simulated Yield Changes')
```

```
# Actual yield changes
sns.heatmap(standardized_yld_chag.corr(), annot=True, cmap='coolwarm', fmt=".
↪1f", ax=axes[1])
axes[1].set_title('Actual Yield Changes')

plt.tight_layout()
plt.show()
```



QUESTION 4

Empirical Analysis of ETFs

QUESTIONS 4(a):

ANSWER

For the group work, we decided to select a utility ETF called XLU, which has about 30 Holdings. The index includes securities of companies from the following industries: electric utilities; water utilities; multi-utilities; independent power and renewable electricity producers; and gas utilities. The fund is non-diversified.

[21]:
```
# Importing requied libraries

import datetime
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import yfinance as yfin
import math
```

```python
from numpy import linalg as LA

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD

from datetime import date

pd.options.display.float_format = "{:,.6f}".format
```

QUESTION 4(b): Get at least 6 months of data (~ 120 data points).

One year of xlu data had been fetched from yfinance database as follows:

[22]:
```python
# Starting and end dates
start = datetime.date(2019, 1, 1)
end = datetime.date(2020, 1, 1)

xlu_tickers = [
    "NEE",   # NextEra Energy Inc.
    "SO",    # Southern Company
    "DUK",   # Duke Energy Corporation
    "AEP",   # American Electric Power Company Inc.
    "D",     # Dominion Energy Inc.
    "SRE",   # Sempra
    "EXC",   # Exelon Corporation
    "VST",   # Vistra Corp.
    "PEG",   # Public Service Enterprise Group Inc.
    "PCG",   # PG&E Corporation
    "XEL",   # Xcel Energy Inc.
    "ED",    # Consolidated Edison Inc.
    "AWK",   # American Water Works Company Inc.
    "WEC",   # WEC Energy Group Inc.
    "EIX",   # Edison International
    "ES",    # Eversource Energy
    "ATO",   # Atmos Energy Corporation
    "CMS",   # CMS Energy Corporation
    "NI",    # NiSource Inc.
    "PNW",   # Pinnacle West Capital Corporation
    "CNP",   # CenterPoint Energy Inc.
    "EVRG",  # Evergy Inc.
    "FE",    # FirstEnergy Corp.
    "NRG",   # NRG Energy Inc.
    "OGE",   # OGE Energy Corp.
    "AEE",   # Ameren Corporation
    "AES",   # The AES Corporation
    "LNT",   # Alliant Energy Corporation
    "UGI",   # UGI Corporation
```

```
    "IDA"    # IDACORP Inc.
]

# Get ETF data
df = yfin.download(xlu_tickers, start, end, auto_adjust = False)["Adj Close"]

# Convert DataFrame index to timezone-aware (UTC)
df.index = df.index.tz_localize('UTC')
```

```
[*********************100%***********************]  30 of 30 completed
```

Let us have a look at the first five rows of the daily data.

[23]:
```
df.head(5)
```

[23]:
```
Ticker                           AEE        AEP        AES        ATO        AWK  \
Date
2019-01-02 00:00:00+00:00  53.270641  58.418915  11.534038  77.131935  79.253159
2019-01-03 00:00:00+00:00  53.404797  58.282764  11.525901  77.715683  79.565666
2019-01-04 00:00:00+00:00  54.159466  58.819359  11.908203  78.711479  80.163826
2019-01-07 00:00:00+00:00  53.538967  58.490982  11.965139  78.136322  79.315689
2019-01-08 00:00:00+00:00  54.385860  59.211826  12.241695  79.157890  80.476295

Ticker                           CMS        CNP          D        DUK         ED  \
Date
2019-01-02 00:00:00+00:00  40.172016  23.255310  54.124611  65.633408  59.839409
2019-01-03 00:00:00+00:00  40.246845  23.388199  53.866074  65.610184  59.990723
2019-01-04 00:00:00+00:00  40.621037  23.820086  54.907814  66.152603  60.962288
2019-01-07 00:00:00+00:00  40.413155  23.911442  54.375546  65.873627  60.078320
2019-01-08 00:00:00+00:00  41.020195  24.368248  55.006657  66.702774  60.444645

Ticker                     …        OGE        PCG        PEG        PNW  \
Date                       …
2019-01-02 00:00:00+00:00  …  28.728683  23.682159  41.137543  63.742741
2019-01-03 00:00:00+00:00  …  28.983988  23.831419  41.129467  64.388840
2019-01-04 00:00:00+00:00  …  29.442028  24.279190  41.574581  65.181107
2019-01-07 00:00:00+00:00  …  29.284349  18.856173  41.663612  65.058037
2019-01-08 00:00:00+00:00  …  29.967651  17.473055  41.825474  66.034920

Ticker                            SO        SRE        UGI        VST        WEC  \
Date
2019-01-02 00:00:00+00:00  33.987125  43.674419  40.206642  19.215071  55.447842
2019-01-03 00:00:00+00:00  34.469093  43.874878  40.706715  19.034771  55.595531
2019-01-04 00:00:00+00:00  34.756718  45.188049  41.591484  19.807493  56.005760
2019-01-07 00:00:00+00:00  34.678986  45.179878  41.676121  20.142340  55.751411
2019-01-08 00:00:00+00:00  35.666275  46.443966  41.991550  20.150928  56.325771

Ticker                           XEL
```

```
Date
2019-01-02 00:00:00+00:00 39.967186
2019-01-03 00:00:00+00:00 39.809719
2019-01-04 00:00:00+00:00 40.199272
2019-01-07 00:00:00+00:00 40.025204
2019-01-08 00:00:00+00:00 40.489357

[5 rows x 30 columns]
```

Overview of the ETF Data

We can use the pandas *describe()* method to show summary stats of our data. We can see that all assets have same number of observations (count) since they all belong to the same portfolio. The other summary stats are relatively basic, like mean and standard deviation along with showing minimum, maximum, and a few quantiles.

[24]:
```
df.describe()
```

[24]:
```
Ticker         AEE         AEP         AES         ATO         AWK         CMS  \
count   252.000000  252.000000  252.000000  252.000000  252.000000  252.000000
mean     62.849828   71.180582   14.031441   90.896079  101.687287   49.038936
std       3.245782    5.109215    1.068596    5.416583    9.813963    3.909191
min      53.270641   58.282764   11.525901   77.131935   79.253159   40.172016
25%      60.837073   67.466379   13.314819   87.135109   93.918928   45.967123
50%      63.895060   72.929676   13.999063   92.650085  104.843285   49.257168
75%      65.094200   75.268436   14.701836   95.203949  110.105621   52.592816
max      68.704857   78.527794   16.838137   99.922478  117.211868   55.235107

Ticker         CNP           D         DUK          ED  …         OGE         PCG  \
count   252.000000  252.000000  252.000000  252.000000  …  252.000000  252.000000
mean     24.547178   59.984373   71.137369   69.934112  …   32.676920   14.893234
std       1.196644    3.548472    2.900220    4.330082  …    1.146570    5.269217
min      21.099731   51.683754   64.827545   59.289909  …   28.728683    3.781185
25%      24.072486   58.064060   69.467724   67.827486  …   32.078154   10.634583
50%      24.733864   59.510866   70.553871   70.953808  …   32.867317   17.030258
75%      25.513127   63.671901   72.968153   72.358799  …   33.288613   18.866123
max      26.322069   66.020332   77.733910   77.404655  …   35.019447   24.279190

Ticker         PEG         PNW          SO         SRE         UGI         VST  \
count   252.000000  252.000000  252.000000  252.000000  252.000000  252.000000
mean     48.428759   72.444122   44.384742   55.560343   39.517781   21.618953
std       2.487902    3.038539    4.653334    5.162168    2.706561    1.422285
min      41.072807   63.435051   33.987125   43.674419   32.437710   18.584152
25%      47.887318   70.596058   40.785181   51.892804   37.717446   20.698181
50%      48.764412   73.282558   44.307463   56.768837   40.448433   21.774376
75%      50.065631   74.517906   49.139084   59.809653   41.575361   22.810292
max      52.458248   77.214256   51.908272   63.737045   43.899559   23.848814
```

```
Ticker         WEC         XEL
count   252.000000  252.000000
mean     70.069637   49.470813
std       6.990331    3.973183
min      55.447842   39.809719
25%      64.181870   46.539076
50%      71.049873   50.785336
75%      76.504229   52.605731
max      81.964592   55.469288

[8 rows x 30 columns]
```

QUESTION 4(C) :

ANSWER Importance of Daily Returns

In financial analysis, the daily return of an asset, XLU-ETF in our case, is a fundamental metric used to measure its performance over a single trading day. It definately aids in quantifying the percentage change in the asset's value between the close of one trading day and the close of the subsequent day. To compute a daily return you have to use the closing prices of the asset on two consecutive trading days.

To account for dividend and stock splits, technically, one must use adjusted closing price to avoid impact on the individual prices of an ETF consitituents assets/holdings.

[25]:
```python
# 1. Compute daily log returns
daily_returns = df.pct_change() # Used simple return to compute the daily return (alternatively: log return can also be used, np.log(df) - np.log(df.shift(1))
daily_returns = daily_returns.dropna()
daily_returns.head()
```

[25]:
```
Ticker                          AEE       AEP       AES       ATO       AWK  \
Date
2019-01-03 00:00:00+00:00  0.002518 -0.002331 -0.000705  0.007568  0.003943
2019-01-04 00:00:00+00:00  0.014131  0.009207  0.033169  0.012813  0.007518
2019-01-07 00:00:00+00:00 -0.011457 -0.005583  0.004781 -0.007307 -0.010580
2019-01-08 00:00:00+00:00  0.015818  0.012324  0.023113  0.013074  0.014633
2019-01-09 00:00:00+00:00 -0.007401 -0.007575 -0.001993 -0.016376 -0.012758

Ticker                          CMS       CNP         D       DUK        ED  \
Date
2019-01-03 00:00:00+00:00  0.001863  0.005714 -0.004777 -0.000354  0.002529
2019-01-04 00:00:00+00:00  0.009297  0.018466  0.019339  0.008267  0.016195
2019-01-07 00:00:00+00:00 -0.005118  0.003835 -0.009694 -0.004217 -0.014500
2019-01-08 00:00:00+00:00  0.015021  0.019104  0.011607  0.012587  0.006097
2019-01-09 00:00:00+00:00 -0.009528 -0.012270 -0.001659 -0.014754 -0.007905

Ticker                     …       OGE       PCG       PEG       PNW  \
```

```
Date                       …
2019-01-03 00:00:00+00:00  …  0.008887  0.006303 -0.000196  0.010136
2019-01-04 00:00:00+00:00  …  0.015803  0.018789  0.010822  0.012304
2019-01-07 00:00:00+00:00  … -0.005356 -0.223361  0.002141 -0.001888
2019-01-08 00:00:00+00:00  …  0.023333 -0.073351  0.003885  0.015016
2019-01-09 00:00:00+00:00  … -0.001897  0.015376 -0.007546 -0.013861

Ticker                           SO       SRE       UGI       VST       WEC  \
Date
2019-01-03 00:00:00+00:00  0.014181  0.004590  0.012438 -0.009383  0.002664
2019-01-04 00:00:00+00:00  0.008344  0.029930  0.021735  0.040595  0.007379
2019-01-07 00:00:00+00:00 -0.002236 -0.000181  0.002035  0.016905 -0.004541
2019-01-08 00:00:00+00:00  0.028469  0.027979  0.007569  0.000426  0.010302
2019-01-09 00:00:00+00:00 -0.008501 -0.009777 -0.010260 -0.006817 -0.005973

Ticker                          XEL
Date
2019-01-03 00:00:00+00:00 -0.003940
2019-01-04 00:00:00+00:00  0.009785
2019-01-07 00:00:00+00:00 -0.004330
2019-01-08 00:00:00+00:00  0.011597
2019-01-09 00:00:00+00:00 -0.007983

[5 rows x 30 columns]
```

Question 4(d): Compute the covariance matrix.

Before standardizing the data, you need to calculate the daily returns of the stocks in the XLU ETF, which is typically done using the percentage change between consecutive closing prices. The covariance matrix is calculated using the daily returns, not the raw prices. It measures how the returns of different XLU ETF stocks move together.

Without calculating the daily returns first, the standardization wouldn't make sense. In fact, standardization makes varying swings (big and small) comparable by adjusting them so they all have an average change of 0 and typical change of 1. This is helpful for analysis where the scale of the data matters.

[26]:
```python
# Standardize stock returns dataset
daily_returns_means = daily_returns.mean()
daily_returns_stds = daily_returns.std()
standardized_returns = (daily_returns - daily_returns_means) /
  ↪daily_returns_stds
standardized_returns.head()
```

[26]:
```
Ticker                          AEE       AEP       AES       ATO       AWK  \
Date
2019-01-03 00:00:00+00:00  0.183450 -0.417105 -0.180855  0.763051  0.275034
2019-01-04 00:00:00+00:00  1.498006  0.948051  2.525074  1.371657  0.663016
2019-01-07 00:00:00+00:00 -1.398540 -0.801923  0.257431 -0.962968 -1.301269
```

```
2019-01-08 00:00:00+00:00  1.688990  1.316904  1.721830  1.401925  1.435249
2019-01-09 00:00:00+00:00 -0.939365 -1.037670 -0.283738 -2.015199 -1.537639

Ticker                          CMS       CNP         D       DUK        ED  \
Date
2019-01-03 00:00:00+00:00  0.074791  0.474443 -0.621066 -0.107618  0.196326
2019-01-04 00:00:00+00:00  0.908490  1.557158  2.049130  0.984212  1.857444
2019-01-07 00:00:00+00:00 -0.707954  0.314894 -1.165504 -0.596891 -1.873469
2019-01-08 00:00:00+00:00  1.550297  1.611336  1.192923  1.531274  0.630099
2019-01-09 00:00:00+00:00 -1.202506 -1.052580 -0.275826 -1.931287 -1.071834

Ticker                     …       OGE       PCG       PEG       PNW  \
Date                       …
2019-01-03 00:00:00+00:00  …  0.936070  0.058488 -0.115138  1.020381
2019-01-04 00:00:00+00:00  …  1.734418  0.196466  1.214921  1.250011
2019-01-07 00:00:00+00:00  … -0.707878 -2.479326  0.167058 -0.252974
2019-01-08 00:00:00+00:00  …  2.603599 -0.821695  0.377514  1.537120
2019-01-09 00:00:00+00:00  … -0.308636  0.158749 -1.002372 -1.520917

Ticker                           SO       SRE       UGI       VST       WEC  \
Date
2019-01-03 00:00:00+00:00  1.412399  0.330930  0.903817 -0.718701  0.143397
2019-01-04 00:00:00+00:00  0.750993  3.088814  1.559423  2.999632  0.675746
2019-01-07 00:00:00+00:00 -0.448059 -0.188285  0.170297  1.237110 -0.670053
2019-01-08 00:00:00+00:00  3.031608  2.876486  0.560491  0.011123  1.005792
2019-01-09 00:00:00+00:00 -1.157927 -1.232719 -0.696680 -0.527805 -0.831618

Ticker                          XEL
Date
2019-01-03 00:00:00+00:00 -0.582390
2019-01-04 00:00:00+00:00  0.959800
2019-01-07 00:00:00+00:00 -0.626237
2019-01-08 00:00:00+00:00  1.163302
2019-01-09 00:00:00+00:00 -1.036720

[5 rows x 30 columns]
```

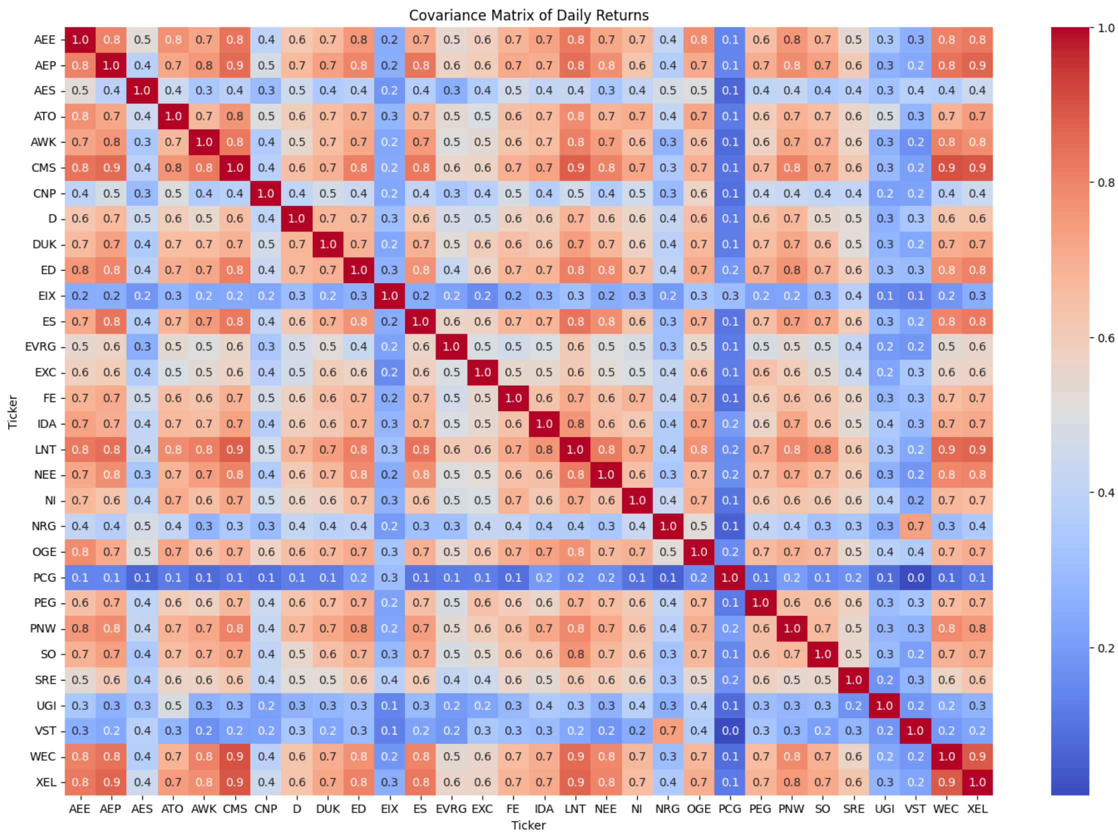
[27]:
```
# Calculate covariance for standardized return matrix
standardized_returns_dvd_sqrt_n=(standardized_returns/math.
 ↪sqrt(len(standardized_returns)-1))
standardized_returns_cov = standardized_returns_dvd_sqrt_n.
 ↪T@standardized_returns_dvd_sqrt_n
standardized_returns_cov.head()
```

[27]:
```
Ticker       AEE       AEP       AES       ATO       AWK       CMS       CNP  \
Ticker
AEE     1.000000  0.806508  0.522383  0.771703  0.724853  0.834286  0.400582
```

```
AEP    0.806508 1.000000 0.389617 0.716671 0.757821 0.851061 0.479604
AES    0.522383 0.389617 1.000000 0.440487 0.343414 0.386723 0.291766
ATO    0.771703 0.716671 0.440487 1.000000 0.665880 0.763396 0.497985
AWK    0.724853 0.757821 0.343414 0.665880 1.000000 0.827253 0.360149

Ticker        D      DUK       ED  …      OGE      PCG      PEG      PNW  \
Ticker                              …
AEE    0.620391 0.662887 0.757679  … 0.781366 0.138427 0.646135 0.758153
AEP    0.661853 0.735293 0.800920  … 0.712110 0.143775 0.699680 0.782281
AES    0.451335 0.354717 0.390818  … 0.505822 0.070451 0.406285 0.421399
ATO    0.592841 0.651968 0.697263  … 0.720000 0.113871 0.600165 0.683688
AWK    0.540264 0.659173 0.713845  … 0.638203 0.082087 0.575091 0.716895

Ticker       SO      SRE      UGI      VST      WEC      XEL
Ticker
AEE    0.689448 0.538975 0.345586 0.258085 0.808670 0.827982
AEP    0.734963 0.604644 0.263486 0.235226 0.840926 0.866046
AES    0.350035 0.378619 0.316272 0.381778 0.376726 0.441356
ATO    0.652024 0.559601 0.485826 0.279782 0.686117 0.742385
AWK    0.691632 0.566794 0.261875 0.164899 0.810270 0.779339

[5 rows x 30 columns]
```

[28]:

```
plt.figure(figsize=(18, 12))
sns.heatmap(standardized_returns_cov, annot=True, cmap='coolwarm', fmt=".1f")
plt.title('Covariance Matrix of Daily Returns')
plt.show()
```



Question 4(e):

(compare and contrast PCA and SVD, explain what the eigenvectors, eigenvalues, singular values etc show us for the specific data, etc,)

ANSWER

Both PCA and SVD are dimensionality reduction technicque. They reduce dimensions of matrices but importantly they do retain substantial information or rather variance of a given financial data.

PCA plays a critical role of identifying key factors behind asset price movements whilst reducing dimensionality of risk-based models. Through PCA, one can establish a diversified portfolio based on the principal components. PCA can keep track of unique patterns and ambiguities in portfolio/ETF, which might not be captured in individual asset.

SVD stands for Singular Value Decomposition. It is a robust factorization technique that can effectively decompose any given matrix (Symmetric or not) into 3 matrices, namely, U, S and V^T. These matrices have mxn, mxn and nxn dimensions respectively. The magnitude of each singular value shows the importance of the corresponding dimension in the data.

Eigen vectors are extracted from PCA and they do represent the principal components which depict relative movements in daily returns of assets. On the other hand, eigenvalues, provides the amount of variance in the daily returns data related to principal components (eigenvector). The higher the eigenvalue, the better it gives information about the asset under analysis.

The following code solutions, shows how Eigenvalues and Eigenvectors help generate Principal Components based on ULX-ETF data under analysis. It is important to note that SVD can be applied to generate eigenvalues and eigenvectors.

[29]:
```
# Calculate eigenvectors and eigenvalues of the covariance matrix of␣
↪standardized dataset
eigenvalues, eigenvectors = np.linalg.eig(standardized_returns_cov)
eigenvalues
```

[29]:
```
array([16.94028088,  1.84467656,  1.26660112,  1.0033836 ,  0.85284607,
        0.78526804,  0.68313883,  0.62148367,  0.56747695,  0.54817649,
        0.50067321,  0.4355354 ,  0.39437057,  0.06999438,  0.08276568,
        0.09744171,  0.10123079,  0.13109828,  0.36516273,  0.35232383,
        0.33001439,  0.29732338,  0.28207821,  0.16274534,  0.17443758,
        0.19353803,  0.24393001,  0.23422616,  0.21796549,  0.21981263])
```

[30]:
```
print(pd.DataFrame(eigenvectors).head())
```

```
         0         1         2         3         4         5         6  \
0 0.213975  0.029617  0.048260 -0.097896  0.130891  0.023761  0.042475
1 0.219064  0.116065  0.034320  0.081349  0.006026 -0.016249  0.048878
2 0.127139 -0.328994  0.058621 -0.160035  0.144884  0.216879  0.387771
3 0.202651 -0.031364 -0.007038 -0.274620 -0.072991  0.025539 -0.076439
4 0.201148  0.183299  0.061812 -0.008566  0.061080  0.105053 -0.072025

         7         8         9  …        20        21        22        23  \
0 -0.207650  0.096307  0.051588  … -0.028883  0.341933 -0.220457 -0.238741
1  0.026485 -0.012477  0.066108  …  0.192387  0.086952 -0.160607  0.223562
2 -0.658986 -0.141501 -0.186631  … -0.012539 -0.139527  0.114488  0.021493
3  0.032491  0.134994  0.117323  … -0.166482  0.374353 -0.255861  0.304626
4  0.013562  0.152696 -0.084538  … -0.322136 -0.388158  0.055149  0.373392

         24        25        26        27        28        29
0 -0.037419  0.028039 -0.020517 -0.062485  0.291977  0.005230
1  0.065085 -0.054074  0.125435  0.118862  0.127297  0.178015
2 -0.085401 -0.007385  0.002403 -0.010940 -0.185881 -0.053974
3  0.157672  0.122336 -0.450374 -0.105967 -0.379816 -0.083053
4 -0.050213  0.051465 -0.031854 -0.431146  0.331905 -0.270027

[5 rows x 30 columns]
```

[31]:
```
# Transform standardized data with Loadings
principal_components = standardized_returns_cov.dot(eigenvectors)
principal_components.columns = ["PC_" + str(i) for i in range(1, 31)]
principal_components.head()
```

[31]:
```
            PC_1      PC_2      PC_3      PC_4      PC_5      PC_6      PC_7  \
Ticker
```

```
AEE    3.624796  0.054633  0.061126 -0.098228  0.111630  0.018659  0.029016
AEP    3.710998  0.214102  0.043469  0.081624  0.005139 -0.012760  0.033391
AES    2.153771 -0.606888  0.074249 -0.160577  0.123564  0.170308  0.264902
ATO    3.432957 -0.057857 -0.008914 -0.275549 -0.062250  0.020055 -0.052218
AWK    3.407511  0.338127  0.078291 -0.008595  0.052092  0.082494 -0.049203

            PC_8      PC_9     PC_10  …     PC_21     PC_22     PC_23  \
Ticker                                …
AEE    -0.129051  0.054652  0.028280  … -0.009532  0.101665 -0.062186
AEP     0.016460 -0.007080  0.036239  …  0.063491  0.025853 -0.045304
AES    -0.409549 -0.080298 -0.102307  … -0.004138 -0.041485  0.032295
ATO     0.020193  0.076606  0.064314  … -0.054941  0.111304 -0.072173
AWK     0.008429  0.086651 -0.046342  … -0.106310 -0.115409  0.015556

           PC_24     PC_25     PC_26     PC_27     PC_28     PC_29     PC_30
Ticker
AEE    -0.038854 -0.006527  0.005427 -0.005005 -0.014636  0.063641  0.001150
AEP     0.036384  0.011353 -0.010465  0.030597  0.027841  0.027746  0.039130
AES     0.003498 -0.014897 -0.001429  0.000586 -0.002562 -0.040516 -0.011864
ATO     0.049576  0.027504  0.023677 -0.109860 -0.024820 -0.082787 -0.018256
AWK     0.060768 -0.008759  0.009960 -0.007770 -0.100986  0.072344 -0.059355

[5 rows x 30 columns]
```

Understanding the explained proportion helps us determine the relative importance of each principal component. By selecting the top principal components with the highest explained proportions, we can reduce the dimensionality of the data while retaining most of the important information.

We used the following codes to demonstrate how to organize the calculated eigenvalues and their corresponding explained proportions into a pandas DataFrame for easier analysis.

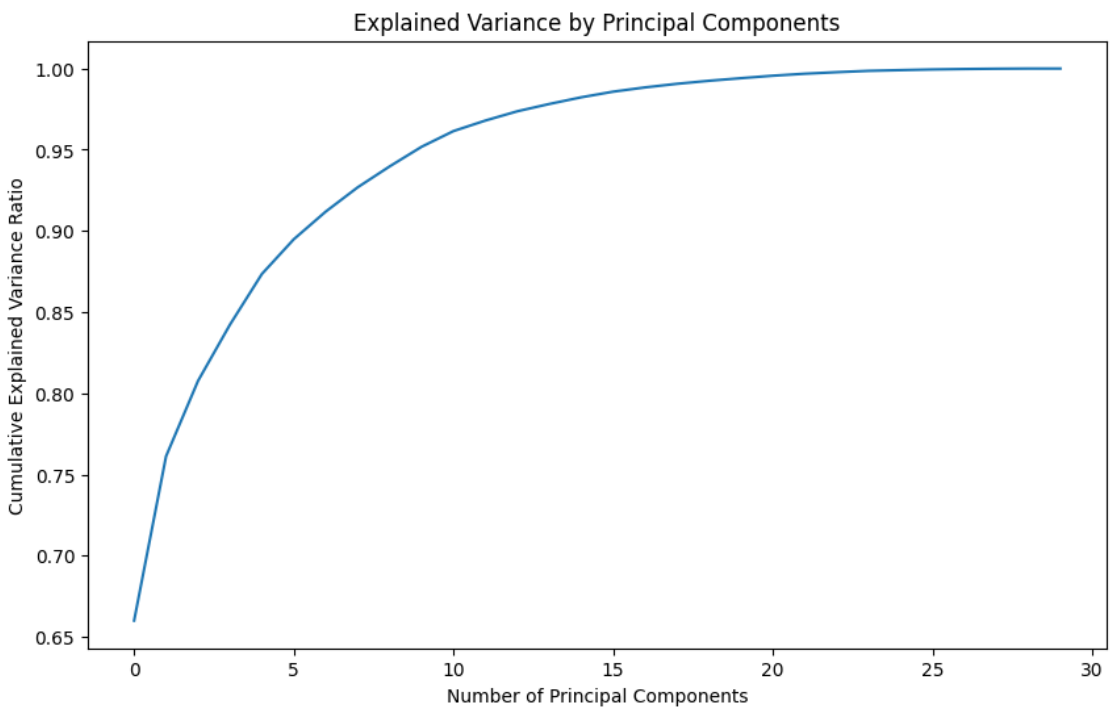
Explained Proportion of 56.47% means that this 'PC_1' alone captures or explains 56.47% of the total variance observed in the standardized daily returns of the XLU ETF's assets.

[32]:
```
# Put data into a DataFrame
df_eigval = pd.DataFrame({"Eigenvalues":eigenvalues}, index=range(1,31))

# Work out explained proportion
df_eigval["Explained proportion"] = df_eigval["Eigenvalues"] / np.
↪sum(df_eigval["Eigenvalues"])
#Format as percentage
df_eigval.style.format({"Explained proportion": "{:.2%}"})
```

[32]:
```
<pandas.io.formats.style.Styler at 0x7d095e4cb810>
```

[33]:
```
# Visualization for PCA

x = standardized_returns_cov.values  # Convert DataFrame to NumPy array
x = StandardScaler().fit_transform(x)  # Standardize the data
```

```
# Apply PCA
pca = PCA(n_components=standardized_returns_cov.shape[1])
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data=principalComponents, columns=['principal␣
 ↪component ' + str(i) for i in range(1, standardized_returns_cov.shape[1] +␣
 ↪1)], index=standardized_returns_cov.index)

# Get explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_

plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(explained_variance_ratio))
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance Ratio')
plt.title('Explained Variance by Principal Components')
plt.show()
```



Question 4(f): Compute the SVD.

The task here involves decomposing the daily returns matrix of the XLU ETF's assets using Singular Value Decomposition (SVD). SVD is a powerful technique for breaking down a matrix into its fundamental components, revealing underlying relationships and patterns within the data.

By applying SVD to the daily returns matrix, we aim to identify the most significant dimensions or factors that drive the variability in the returns of the XLU ETF's assets. Secondly, gain insights into the relationships between different assets within the ETF and how they contribute to overall portfolio risk and return and transform the original data into a more manageable form, making it easier to interpret and model.

[34]:
```python
# Perform SVD for stock returns
U, s, VT = np.linalg.svd(daily_returns)
```

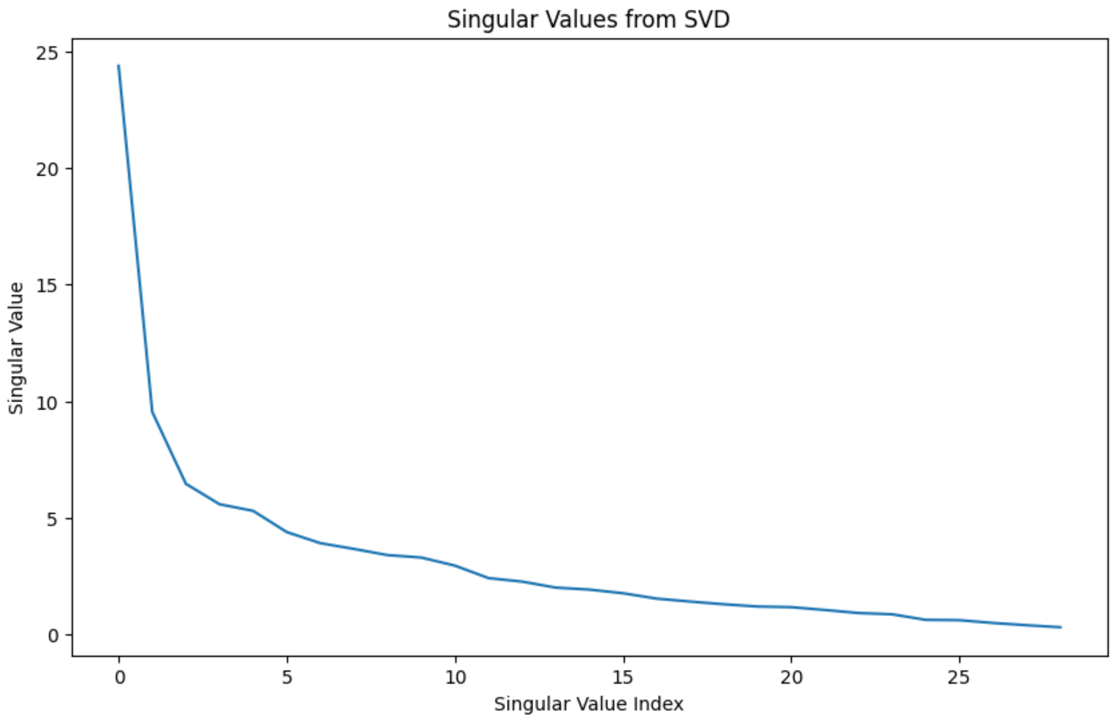
[36]:
```python
# Use SVD to calculate eigenvectors and eigenvalues of the covariance matrix of␣
↪standardized returns
U_st_return, s_st_return, VT_st_return = np.linalg.
↪svd(standardized_returns_dvd_sqrt_n)
print("\nSquared Singular values (eigenvalues):")
print(s_st_return**2)
print("\nMatrix V (eigenvectors)")
print(pd.DataFrame(VT_st_return.T).head())
```

```
Squared Singular values (eigenvalues):
[16.94028088  1.84467656  1.26660112  1.0033836   0.85284607  0.78526804
  0.68313883  0.62148367  0.56747695  0.54817649  0.50067321  0.4355354
  0.39437057  0.36516273  0.35232383  0.33001439  0.29732338  0.28207821
  0.24393001  0.23422616  0.21981263  0.21796549  0.19353803  0.17443758
  0.16274534  0.13109828  0.10123079  0.09744171  0.08276568  0.06999438]

Matrix V (eigenvectors)
         0         1         2         3         4         5         6   \
0 -0.213975 -0.029617  0.048260 -0.097896  0.130891 -0.023761  0.042475
1 -0.219064 -0.116065  0.034320  0.081349  0.006026  0.016249  0.048878
2 -0.127139  0.328994  0.058621 -0.160035  0.144884 -0.216879  0.387771
3 -0.202651  0.031364 -0.007038 -0.274620 -0.072991 -0.025539 -0.076439
4 -0.201148 -0.183299  0.061812 -0.008566  0.061080 -0.105053 -0.072025

         7         8         9   …        20        21        22        23  \
0 -0.207650  0.096307  0.051588  … -0.005230 -0.291977  0.028039  0.037419
1  0.026485 -0.012477  0.066108  … -0.178015 -0.127297 -0.054074 -0.065085
2 -0.658986 -0.141501 -0.186631  …  0.053974  0.185881 -0.007385  0.085401
3  0.032491  0.134994  0.117323  …  0.083053  0.379816  0.122336 -0.157672
4  0.013562  0.152696 -0.084538  …  0.270027 -0.331905  0.051465  0.050213

         24        25        26        27        28        29
0  0.238741 -0.248865 -0.007800  0.621952  0.029046 -0.106701
1 -0.223562 -0.695037  0.198811 -0.371491  0.046585  0.098040
2 -0.021493 -0.033900  0.006939 -0.078475  0.070041  0.066776
3 -0.304626  0.069615 -0.078672 -0.166916  0.101930 -0.138801
4 -0.373392 -0.089393 -0.085101  0.056272 -0.084307 -0.087949

[5 rows x 30 columns]
```

[37]:
```
# Presenting the result
print("ETF Returns Matrix Dimension:")
print(daily_returns.shape)
print("\nDimension of Matrix U:")
print(U_st_return.shape)
print("\nSingular values:")
print(s_st_return**2)
print("\nDimension of Matrix V^T:")
print(VT_st_return.shape)
```

```
ETF Returns Matrix Dimension:
(251, 30)

Dimension of Matrix U:
(251, 251)

Singular values:
[16.94028088  1.84467656  1.26660112  1.0033836   0.85284607  0.78526804
  0.68313883  0.62148367  0.56747695  0.54817649  0.50067321  0.4355354
  0.39437057  0.36516273  0.35232383  0.33001439  0.29732338  0.28207821
  0.24393001  0.23422616  0.21981263  0.21796549  0.19353803  0.17443758
  0.16274534  0.13109828  0.10123079  0.09744171  0.08276568  0.06999438]

Dimension of Matrix V^T:
(30, 30)
```

[38]:
```
# Visualization for SVD
svd = TruncatedSVD(n_components=standardized_returns_cov.shape[1] - 1)
svd_components = svd.fit_transform(x)
svdDf = pd.DataFrame(data=svd_components, columns=['singular value ' + str(i)␣
 ↪for i in range(1, standardized_returns_cov.shape[1])],␣
 ↪index=standardized_returns_cov.index)

plt.figure(figsize=(10, 6))
plt.plot(svd.singular_values_)
plt.xlabel('Singular Value Index')
plt.ylabel('Singular Value')
plt.title('Singular Values from SVD')
plt.show()
```


Singular Values from SVD
Singular Value
Singular Value Index
25
20
15
10
5
0
0
5
10
15
20
25