Python Labs

Yield Curve PCA Decomposition

Created Kannan Singaravelu. Edited and PCA Projection amended by Dr Richard Diamond

kannan.singaravelu@fitchlearning.com

Dimensionality Reduction

One of the main difficulties in today's environment is being able to visualize data easily. There is too much information, too much news, and too much data. Dimensionality is the number of dimensions, features or input variables associated in a dataset and dimensionality reduction means reducing the number of features in a dataset.

Dimensionality reduction algorithms project high-dimensional data to a low-dimensional space while retaining as much of the variation as possible. There are two main approaches to dimensionality reduction.

- The first one is known as linear projection which involves linearly projecting data from a high-dimensional space to a low-dimensional space. This includes techniques such as principal component analysis (PCA).
- The second approach is known as manifold learning which is also referred to as nonlinear dimensionality reduction. This includes techniques such as Uniform manifold approximation and projection (UMAP).

Dimensionality reduction techniques help to address the curse of dimensionality.

Principal Component

PCA is a linear dimensionality reduction techniqu where the algorithm finds a low-dimensional representation of the data while retaining as much of the variation as possible and help reduce the complexity.

The main concept behind the PCA is to consider the correlation among features. If the correlation is very high among a subset of the features, PCA will attempt to combine the highly correlated features and represent this data with a smaller number of linearly uncorrelated features. The algorithm keeps performing this correlation reduction, finding the directions of maximum variance in the original high-dimensional data and projecting them onto a smaller dimensional space. These newly derived components are known as **principal components**.

Investors often refer to movements in the yield curve in terms of three driving factors:

- Level
- Slope
- Curvature

PCA formalizes this viewpoint and allows us to evaluate when a segment of the yield curve has cheapened or richened beyond that prescribed by recent yield movements. The essence of PCA in the context of rates market is that most yield curve movements can be represented as a set of two to three independent driving factors – the principal components (PCs) – along with their relative weightings. And, with these components, it is possible to reconstruct the original features.

We'll apply PCA to the set of yield curves fitted using the HJM model as discussed during the lecture. The PCs are ordered so that the first PC is the most important in capturing variability in the yield curves, the second PC is next most important, and so on.

The most intuitive way of obtaining PCs is via eigenvalue decomposition of a covariance matrix. The covariance measures the central tendency and talks about deviation from the mean. Intuitively, PCs represent ways in which the forward rates making up a yield curve can deviate from their mean levels.

Load Libraries

In [ ]:
```
# Import libraries
import numpy as np
import pandas as pd
import os as os

# Cufflinks library allows direct plotting of Plotly interactive charts f
import cufflinks as cf
cf.set_config_file(offline=True)

# Heatmap of covariance matrix
import plotly.graph_objs as go
from plotly.subplots import make_subplots

# scikit
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

In [ ]:
```
pd.set_option('display.max_rows', 5000)
pd.set_option('display.max_columns', 100)
pd.set_option('display.width', 1000)
```

Datasets

Swap Curve DAILY 2002 to 2007

```
- hjm-pca.csv has DAILY forward curves for Bank Liability
Curve (BLC), historically derived by the BOE from the
borrowings of AAA to AA-rated financial institutions.
The curve was known as LIBOR Curve, but
professional/Bloomberg data to use usually called Swap
Curve.
Period is approximately five years from January 2007 (top
lines) to January 2002 (bottom lines).
```

Gilts Curve MONTHLY 1970 to 2015

```
- gilts_spot_1970-2015.xlsx has MONTHLY spot curves for
Government Liability Curve (GLC), stripped from UK
Treasury Gilts. Excel sheet "4. spot curve"
Period is much longer from January 1970 (now the oldest
curves in top lines) to December 2015.
```

In [ ]:
```
# Check working directory
os.getcwd()

# Set working directory if necessary
# work_dir = "INSERT PATH TO FILE LOCATION"
# os.chdir(work_dir)
```

In [ ]:
```
data = pd.read_csv('./data/hjm_pca_2002-07.csv', index_col=0, sep ='\t')
```

In [ ]:
```
data.head()
```

Out[ ]:

	0.08	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0	6.5	7.0
1	5.77	6.44	6.71	6.65	6.50	6.33	6.15	5.99	5.84	5.71	5.57	5.44	5.30	5.16	5.01
2	5.77	6.45	6.75	6.68	6.54	6.39	6.23	6.08	5.95	5.82	5.69	5.56	5.43	5.28	5.13
3	5.78	6.44	6.74	6.68	6.56	6.41	6.26	6.12	5.98	5.84	5.71	5.57	5.43	5.28	5.12
4	5.74	6.41	6.69	6.62	6.49	6.35	6.20	6.06	5.93	5.79	5.66	5.52	5.38	5.23	5.07
5	5.74	6.40	6.64	6.55	6.42	6.27	6.13	5.98	5.85	5.72	5.58	5.44	5.30	5.15	5.00

In [ ]:
```
data.shape
```

Out[ ]:
```
(1264, 51)
```

Rows x Columns

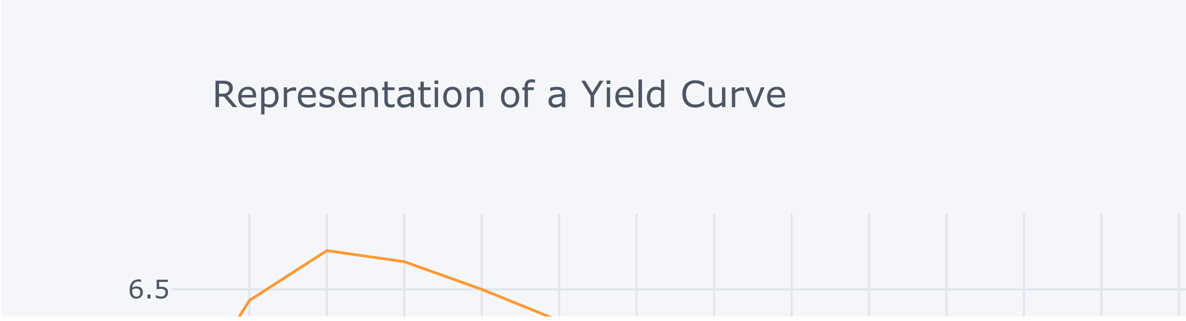
Representation of a yield curve as 50 forward rates. As the yield curve evolves over time, each forward rate can change. It is understood that adjacent points on the yield curve do not move independently. PCA is a method for identifying the dominant ways in which various points on the yield curve move together.

PCA allows us to take a set of yield curves, and analyse their movements in the model-free approach -- it is **unsupervised learning**.

The reduced model for the yield curve retains only a small number of principal components (PCs) -- typically 3 to 5 eigenvectors depending on amount of variance covered. That model can reproduce the vast majority of yield curves. That reduced model has fewer sources of uncertainty (i.e. dimensions) compared to 51 tenors of the original the yield curve, particularly that we know they don't move fully independently.

Plot Curves

In [ ]:
```
# Plot curve
data.iloc[0].iplot(title = 'Representation of a Yield Curve')
```



In [ ]:
```
# Plot all curves
data.T.iplot(title='Daily Yield Curves')
```

We'll now produce the volatility chart by taking the first difference (scaling) and calculating historical variance by each individual maturity.

In [ ]:
```
diff_ = data.diff(-1)
diff_.dropna(inplace=True)
```

In [ ]:
```
diff_.tail()
```

Out[ ]:

	0.08	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0
1259	0.00	0.03	0.04	0.03	0.02	0.02	0.01	0.01	0.00	0.00	0.00	0.00	-0.01
1260	0.02	0.01	0.00	0.00	0.00	-0.01	-0.01	-0.01	0.00	-0.01	-0.01	-0.01	0.00
1261	-0.01	-0.03	-0.08	-0.12	-0.13	-0.13	-0.13	-0.13	-0.14	-0.13	-0.14	-0.14	-0.14
1262	0.00	0.00	0.01	0.02	0.01	0.01	0.01	0.00	0.01	0.00	0.00	0.00	0.00
1263	0.02	0.00	0.03	0.03	0.04	0.04	0.05	0.06	0.06	0.06	0.07	0.07	0.06

In [ ]:
```
diff_.shape
```

Out[ ]:
```
(1263, 51)
```

Derive Volatility

The drift of forward rate is determined by volatility of forward rates -- we have learned this in the HJM Model drift function m(), which is vol times integral of vol.

For general knowledge, we compute the volatility at different tenors of the curve (HJM PCA curve data.)

In [ ]:
```
vol = np.std(diff_, axis=0) * 10000
```

In [ ]:
```
vol[:21].iplot(title='Volatility of Daily Yields', xTitle='Tenor', yTitle
                color='cornflowerblue')
```

The above volatility plot is of the averaged values, but we can see that different parts of the yield curve move differently. As you can see volatility is very significant, especially at the shorter end of the curve. This means that 1-year and 2-year rates seems to move up and down a lot as compared to other tenors.

It is never all up or all down and PCA help us figure out exactly what is going. Covariance of daily changes shows dependency of different rates. Principal components can be calculated by finding the eigenvalues and eigenvectors of this covariance matrix of below.

Compute Covariance

In [ ]:
```
cov_= pd.DataFrame(np.cov(diff_, rowvar=False)*252/10000,
                   columns=diff_.columns, index=diff_.columns)

cov_.style.format("{:.4%}")
```

Out[ ]:

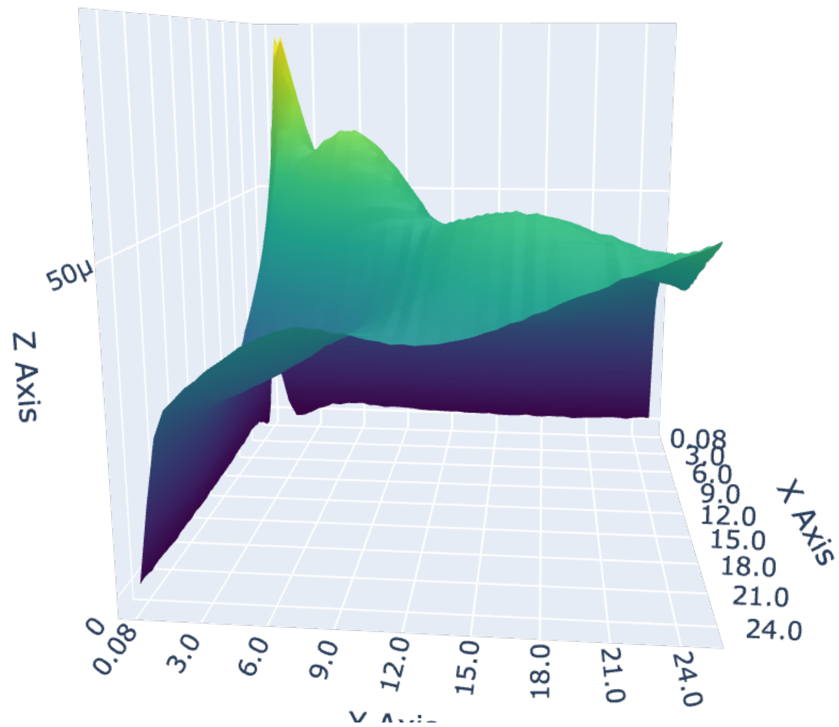
	0.08	0.5	1.0	1.5	2.0	2.5	3.0	3.5
0.08	0.0040%	0.0009%	0.0002%	-0.0001%	-0.0001%	-0.0000%	0.0001%	0.0001%
0.5	0.0009%	0.0063%	0.0055%	0.0041%	0.0035%	0.0033%	0.0031%	0.0029%
1.0	0.0002%	0.0055%	0.0082%	0.0077%	0.0068%	0.0061%	0.0056%	0.0052%
1.5	-0.0001%	0.0041%	0.0077%	0.0082%	0.0075%	0.0069%	0.0063%	0.0058%
2.0	-0.0001%	0.0035%	0.0068%	0.0075%	0.0072%	0.0067%	0.0063%	0.0059%
2.5	-0.0000%	0.0033%	0.0061%	0.0069%	0.0067%	0.0065%	0.0062%	0.0060%
3.0	0.0001%	0.0031%	0.0056%	0.0063%	0.0063%	0.0062%	0.0061%	0.0060%
3.5	0.0001%	0.0029%	0.0052%	0.0058%	0.0059%	0.0060%	0.0060%	0.0060%
4.0	0.0002%	0.0028%	0.0048%	0.0055%	0.0056%	0.0058%	0.0058%	0.0059%
4.5	0.0002%	0.0027%	0.0045%	0.0051%	0.0054%	0.0055%	0.0057%	0.0058%
5.0	0.0002%	0.0026%	0.0042%	0.0049%	0.0051%	0.0054%	0.0056%	0.0058%
5.5	0.0002%	0.0025%	0.0040%	0.0046%	0.0049%	0.0052%	0.0054%	0.0057%
6.0	0.0002%	0.0024%	0.0038%	0.0044%	0.0047%	0.0051%	0.0053%	0.0056%
6.5	0.0002%	0.0022%	0.0036%	0.0042%	0.0046%	0.0049%	0.0052%	0.0055%
7.0	0.0002%	0.0021%	0.0035%	0.0041%	0.0044%	0.0048%	0.0051%	0.0054%
7.5	0.0002%	0.0020%	0.0033%	0.0039%	0.0043%	0.0046%	0.0049%	0.0052%
8.0	0.0002%	0.0019%	0.0032%	0.0038%	0.0041%	0.0044%	0.0047%	0.0050%
8.5	0.0002%	0.0018%	0.0031%	0.0036%	0.0039%	0.0042%	0.0045%	0.0048%
9.0	0.0002%	0.0017%	0.0029%	0.0035%	0.0038%	0.0041%	0.0043%	0.0046%
9.5	0.0002%	0.0016%	0.0028%	0.0034%	0.0036%	0.0039%	0.0042%	0.0044%

10.0	0.0001%	0.0015%	0.0027%	0.0032%	0.0035%	0.0037%	0.0040%	0.0042%
10.5	0.0001%	0.0014%	0.0026%	0.0031%	0.0033%	0.0035%	0.0037%	0.0040%
11.0	0.0001%	0.0013%	0.0025%	0.0029%	0.0031%	0.0034%	0.0036%	0.0038%
11.5	0.0001%	0.0012%	0.0023%	0.0028%	0.0030%	0.0032%	0.0033%	0.0035%
12.0	0.0001%	0.0011%	0.0022%	0.0027%	0.0029%	0.0030%	0.0032%	0.0033%
12.5	0.0001%	0.0011%	0.0021%	0.0026%	0.0027%	0.0029%	0.0030%	0.0032%
13.0	0.0001%	0.0010%	0.0020%	0.0025%	0.0026%	0.0028%	0.0029%	0.0030%
13.5	0.0001%	0.0009%	0.0020%	0.0024%	0.0025%	0.0027%	0.0028%	0.0029%
14.0	0.0001%	0.0009%	0.0019%	0.0023%	0.0024%	0.0026%	0.0026%	0.0028%
14.5	0.0001%	0.0008%	0.0018%	0.0022%	0.0023%	0.0025%	0.0025%	0.0026%
15.0	0.0001%	0.0008%	0.0018%	0.0022%	0.0023%	0.0024%	0.0025%	0.0026%
15.5	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0023%	0.0024%	0.0025%
16.0	0.0001%	0.0007%	0.0017%	0.0021%	0.0022%	0.0023%	0.0024%	0.0024%
16.5	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0023%	0.0024%	0.0025%
17.0	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0023%	0.0023%	0.0024%
17.5	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0023%	0.0024%	0.0025%
18.0	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0023%	0.0024%	0.0024%
18.5	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0023%	0.0024%	0.0025%
19.0	0.0001%	0.0008%	0.0017%	0.0021%	0.0022%	0.0024%	0.0024%	0.0025%
19.5	0.0001%	0.0009%	0.0018%	0.0022%	0.0023%	0.0024%	0.0025%	0.0026%
20.0	0.0000%	0.0009%	0.0018%	0.0022%	0.0023%	0.0024%	0.0025%	0.0026%
20.5	0.0001%	0.0009%	0.0018%	0.0022%	0.0024%	0.0025%	0.0026%	0.0027%
21.0	0.0001%	0.0010%	0.0019%	0.0023%	0.0025%	0.0026%	0.0027%	0.0028%
21.5	0.0001%	0.0010%	0.0019%	0.0023%	0.0025%	0.0026%	0.0027%	0.0028%
22.0	0.0001%	0.0010%	0.0020%	0.0025%	0.0026%	0.0027%	0.0028%	0.0029%
22.5	0.0001%	0.0011%	0.0021%	0.0025%	0.0026%	0.0028%	0.0029%	0.0030%
23.0	0.0001%	0.0012%	0.0021%	0.0026%	0.0027%	0.0028%	0.0029%	0.0031%
23.5	0.0001%	0.0012%	0.0022%	0.0026%	0.0028%	0.0029%	0.0030%	0.0032%
24.0	0.0001%	0.0012%	0.0022%	0.0027%	0.0028%	0.0030%	0.0031%	0.0032%
24.5	0.0001%	0.0013%	0.0023%	0.0027%	0.0029%	0.0031%	0.0032%	0.0033%
25.0	0.0001%	0.0013%	0.0024%	0.0028%	0.0030%	0.0032%	0.0033%	0.0034%

In [ ]:
```
# Heatmap appropriate for Covariance Matrix
fig_matrix = go.Figure(data=go.Heatmap(z=cov_, colorscale='Viridis'))
fig_matrix.update_layout(title='Covariance Matrix Heatmap')
fig_matrix.show()
```

In [ ]:
```
# 3D Surface Plot with larger dimensions
x, y = np.meshgrid(cov_.columns, cov_.index)
fig_surface = make_subplots(rows=1, cols=1, specs=[[{'type': 'surface'}]]
fig_surface.add_trace(go.Surface(z=cov_.values, x=x, y=y, colorscale='Vir

# Update layout for larger dimensions
fig_surface.update_layout(title='Covariance 3D Surface Plot (rotate)',
                          scene=dict(
                              xaxis=dict(title='X Axis'),
                              yaxis=dict(title='Y Axis'),
                              zaxis=dict(title='Z Axis'),
                          ),
                          width=800,  # Adjust width as needed
                          height=600  # Adjust height as needed
                          )

# Show the plot
fig_surface.show()

# Observation: if we remove the 0.08 tenor (where covariance peaks), we a
```

Covariance 3D Surface Plot



Singular Value Decomposition

In [ ]:
```
eigenvalues, eigenvectors = np.linalg.eig(cov_)

# Sort values (good practice)
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:,idx]

# Format into DataFrame
df_eigval = pd.DataFrame({"Eigenvalues": eigenvalues})
```

In [ ]:
```
df_eigval.head()
```

Out[ ]:

	Eigenvalues
0	0.002029
1	0.000463
2	0.000163
3	0.000085
4	0.000051

Explained Variance R^2 as sum of eigenvalues

In [ ]:

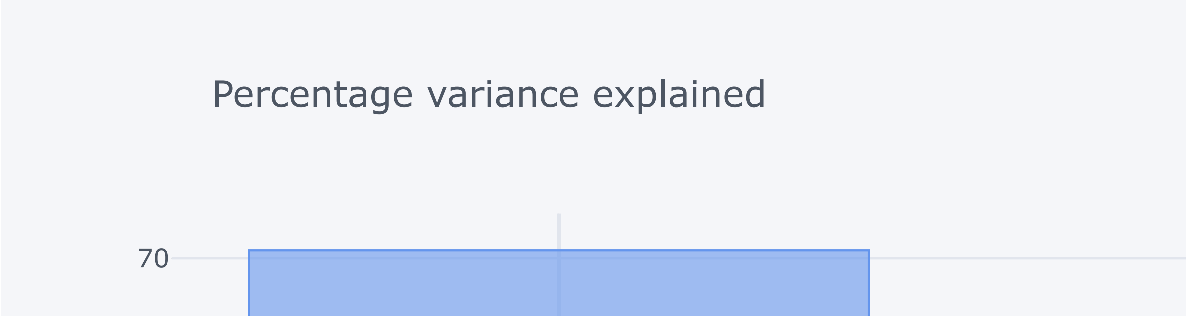
```
# Work out explained proportion
df_eigval["Var Explained"] = df_eigval["Eigenvalues"] / np.sum(df_eigval[
df_eigval = df_eigval[:8]

#Format as percentage
df_eigval.style.format({"Var Explained": "{:.2%}"})
```

Out[ ]:

	Eigenvalues	Var Explained
0	0.002029	70.81%
1	0.000463	16.17%
2	0.000163	5.70%
3	0.000085	2.97%
4	0.000051	1.78%
5	0.000033	1.16%
6	0.000016	0.55%
7	0.000004	0.16%

In [ ]:

```
(df_eigval['Var Explained'][:5]*100).iplot(kind='bar', title='Percentage
                                            color='cornflowerblue')
```



Visualize PCs

In [ ]:
```
# Subsume first 3 components into a dataframe
pcadf = pd.DataFrame(eigenvectors[:,0:3], columns=['e1','e2','e3'], index
pcadf[:10]
```

Out[ ]:

	e1	e2	e3
0.08	0.004091	-0.008275	0.000235
0.5	0.056204	-0.161934	-0.271539
1.0	0.101034	-0.239236	-0.401805
1.5	0.116817	-0.243675	-0.357226
2.0	0.121388	-0.235475	-0.275176
2.5	0.125890	-0.226757	-0.195816
3.0	0.129107	-0.219537	-0.123907
3.5	0.133088	-0.211509	-0.062428
4.0	0.136317	-0.204675	-0.007698
4.5	0.139725	-0.197136	0.041132

In [ ]:

```
pcadf.iplot(title='Principal Components for Forward Curve (HJM Lecture) U
            yTitle='change in yield (bps)')
```

One of the key interpretations of PCA as applied to interest rates are the components of the yield curve. We can attribute the first three principal components to

- Parallel shifts in yield curve (shifts across the entire yield curve)
- Changes in short/long rates (steepening/flattening of the curve)
- Changes in curvature of the model (twists)

The first PC represents the situation that all forward rates in the yield curve move in the same direction but points around the 15 year term move more than points at the shorter or longer parts of the yield curve. This corresponds to a general rise (or fall) of all of the forward rates in the yield curve, but cannot be called a uniform or parallel shift. The impact of the first PC can be easily observed amongst the yield curves as it contributes more than 71% of the variability.

The second PC represents situations in which the short end of the yield curve moves up at the same time as the long end moves down, or vice versa. This is often described as a tilt in the yield curve, although in practice there is more subtle definition to the shape. This reflects the particular yield curves that were used for the analysis, as well as the structural model and calibration that were used to create them. In this excample, the influence of the second PC accounts for about 16% of the variability in the yield curves.

The third PC is further interpreted as a higher order buckling in which the short end and long end move up at the same time as a region of medium term rates move down, or vice versa. In this particular example, this type of movement is only responsible for about 5.70% of the variability.

Having identified the most important factors, we can use their functional form to predict the most likely evolution of the yeild curve. Thus, a simple linear regression is fitted for the shift factor as it simply moves the curve up and down. Second degree polynomial is fitted for the tilt factor and higher degree can approximate flexing. Thus, yield curve can be approximated by linear combination of first three loadings.

UK Gilts - Nominal Spot

The purpose of applying PCA to financial markets is to explain the price changes of different assets through a smaller set of factors. This is achieved via the dimensionality reduction of the observations where we pick meaningful factors (among many) explaining the most of the price changes. We'll now apply the principal component analysis to UK government bond spot rates [1] from 0.5 years up to 10 years to maturity.

We will perform Singular Value Decomposition (SVD) of covariance matrix using two Python functionalities: *numpy.linalg* and *sklearn.PCA*.

We will remember to scale the data in both implementations.

Gilts Curve MONTHLY 1970 to 2015

```
- gilts_spot_1970-2015.xlsx has MONTHLY spot curves for
Government Liability Curve (GLC), stripped from UK
Treasury Gilts. Excel sheet "4. spot curve"
Period is much longer from January 1970 (now the oldest
curves in top lines) to December 2015;

- we limit our analysis to $[1Y, 10.5Y]$ chunk of the spot
give. The likely implication is we end up with limited
usefulness of PCA and a very strong PC1;

- looking into data, the first column at tenor 0.5 has a
lot of missing values. With MONTHLY frequency that would
be lot of monthly curves thrown out of analysis, and
particularly as we cut the curve to the front end.
```

In [ ]:
```
# Import Curve Data
df = pd.read_excel("./data/gilts_spot_1970-2015.xlsx", index_col=0, heade

# IMPORTANT DATA PROCESSING
# Limit curve to 10.5Y tenor.  Semi-annual increments give 20 columns
# Tenor 0.5Y would have given a lot missed values, so the entire monthy c

df = df.iloc[:, 1:21]

df.head()
```

Out[ ]:

years:	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5
1970-01-31	8.635354	8.707430	8.700727	8.664049	8.618702	8.572477	8.528372	8.487617
1970-02-28	8.413131	8.397269	8.370748	8.337633	8.301590	8.265403	8.230804	8.198713
1970-03-31	7.744187	7.782761	7.795017	7.793104	7.784963	7.775288	7.766459	7.759564
1970-04-30	7.606512	7.864352	7.973522	8.002442	7.992813	7.967524	7.938335	7.911422
1970-05-31	7.391107	7.735838	7.862182	7.877510	7.840673	7.782249	7.718053	7.656856

In [ ]:

```
df = df.dropna(how="any")
df.shape
```

Out[ ]:

```
(550, 20)
```

In [ ]:

```
# StandardScaler() by defaults normalises data -- computes Z-scores by co
scaler = StandardScaler()
scaler.fit(df)

df1 = pd.DataFrame(scaler.transform(df))
df1.head()
```

Out[ ]:

	0	1	2	3	4	5	6	7
0	0.438865	0.440632	0.418813	0.390185	0.360774	0.332320	0.305297	0.279832
1	0.381957	0.360305	0.332602	0.304217	0.276632	0.250291	0.225334	0.201798
2	0.210651	0.201157	0.182185	0.160805	0.139552	0.119366	0.100553	0.083182
3	0.175394	0.222287	0.228822	0.215938	0.194702	0.170718	0.146740	0.124200
4	0.120232	0.189004	0.199733	0.183035	0.154334	0.121225	0.087545	0.055440

Covariance Matrix (Scaled Data)

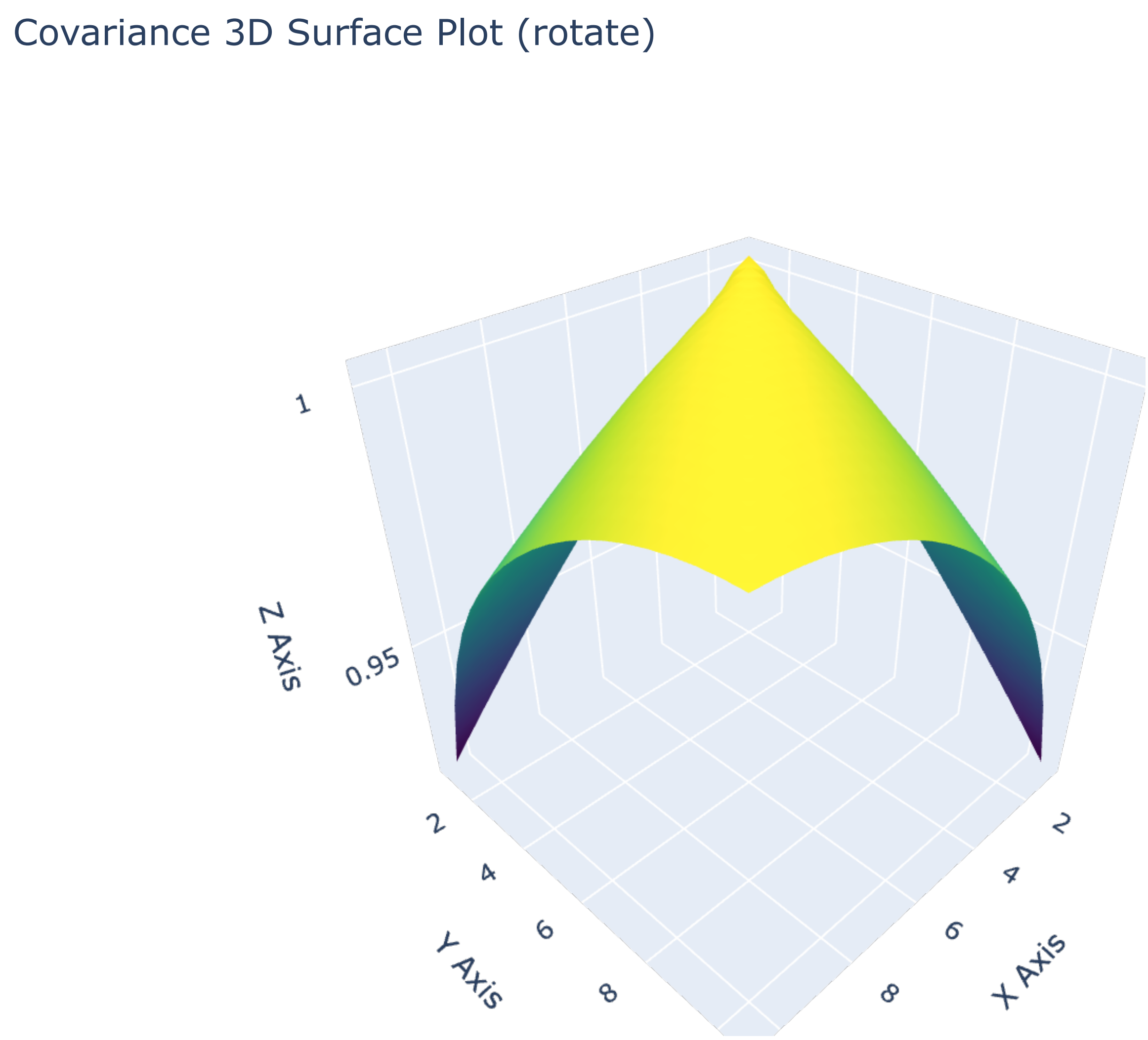
In [ ]:

```
cov_array = np.cov(df1, rowvar=False)
# cov_df1 = pd.DataFrame(cov_array) #, index=range(1,21), columns=range(1

cov_df1 = pd.DataFrame(cov_array, columns=df.columns, index=df.columns)
cov_df1 .style.format("{:.4}")
```

Out[ ]:

	1.000000	1.500000	2.000000	2.500000	3.000000	3.500000	4.000000
1.000000	1.002	0.9998	0.9958	0.9912	0.9866	0.9822	0.9779
1.500000	0.9998	1.002	1.001	0.9982	0.9951	0.9919	0.9886
2.000000	0.9958	1.001	1.002	1.001	0.9994	0.9973	0.995
2.500000	0.9912	0.9982	1.001	1.002	1.001	1.0	0.9986
3.000000	0.9866	0.9951	0.9994	1.001	1.002	1.001	1.001
3.500000	0.9822	0.9919	0.9973	1.0	1.001	1.002	1.002
4.000000	0.9779	0.9886	0.995	0.9986	1.001	1.002	1.002
4.500000	0.9739	0.9853	0.9924	0.9967	0.9993	1.001	1.002
5.000000	0.9699	0.982	0.9897	0.9946	0.9977	0.9997	1.001
5.500000	0.9659	0.9786	0.9869	0.9923	0.9959	0.9983	1.0
6.000000	0.9619	0.9751	0.9839	0.9898	0.9938	0.9966	0.9987
6.500000	0.9578	0.9716	0.9808	0.9871	0.9915	0.9947	0.9971
7.000000	0.9537	0.9679	0.9775	0.9842	0.989	0.9925	0.9953
7.500000	0.9494	0.9641	0.9742	0.9812	0.9863	0.9902	0.9932
8.000000	0.9451	0.9602	0.9706	0.978	0.9834	0.9876	0.991
8.500000	0.9407	0.9562	0.967	0.9747	0.9804	0.9849	0.9886
9.000000	0.9362	0.952	0.9632	0.9712	0.9773	0.982	0.986
9.500000	0.9315	0.9478	0.9593	0.9677	0.974	0.979	0.9832
10.000000	0.9267	0.9434	0.9553	0.9639	0.9705	0.9759	0.9803
10.500000	0.9218	0.9389	0.9511	0.9601	0.967	0.9725	0.9772

In [ ]:
```
# 3D Surface Plot with larger dimensions
x, y = np.meshgrid(cov_df1.columns, cov_df1.index)
fig_surface = make_subplots(rows=1, cols=1, specs=[[{'type': 'surface'}]]
fig_surface.add_trace(go.Surface(z=cov_df1.values, x=x, y=y, colorscale='

# Update layout for larger dimensions
fig_surface.update_layout(title='Covariance 3D Surface Plot (rotate)',
                          scene=dict(
                              xaxis=dict(title='X Axis'),
                              yaxis=dict(title='Y Axis'),
                              zaxis=dict(title='Z Axis'),
                          ),
                          width=800,  # Adjust width as needed
                          height=600  # Adjust height as needed
                          )

# Show the plot
fig_surface.show()

# Observation: we have ended up with very robust covariance matrix, devoi
```



Singular Value Decomposition

In [ ]:
```
eigenvalues, eigenvectors = np.linalg.eig(cov_array)

# Sort values (good practice)
idx = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:,idx]

# Format into DataFrame (but we output array type below -- to show how sm
df1_eigval = pd.DataFrame({"Eigenvalues": eigenvalues}) #, index=range(1,
```

In [ ]:
```
eigenvalues
```

Out[ ]:
```
array([1.97536839e+01, 2.63618514e-01, 1.66472447e-02, 2.09709989e-03,
       3.61048910e-04, 1.98613387e-05, 2.05085861e-06, 1.57775294e-07,
       2.18653003e-08, 5.24252240e-09, 1.03925795e-09, 2.41742911e-10,
       6.16969240e-11, 1.56642017e-11, 6.87563072e-12, 2.16115375e-12,
       7.60439950e-13, 2.01826232e-13, 3.25176662e-14, 7.22731909e-15])
```

In [ ]:
```
# Format into a DataFrame
df1_eigvec = pd.DataFrame(eigenvectors) #, index=range(1,21))

eigenvectors[:,0] # Only PC1 is of relevance
```

Out[ ]:
```
array([0.218112  , 0.22071219, 0.22234786, 0.22337726, 0.22404304,
       0.22448583, 0.22478226, 0.2249718 , 0.225074  , 0.22509903,
       0.22505332, 0.22494209, 0.22477012, 0.22454186, 0.22426118,
       0.2239312 , 0.22355415, 0.22313142, 0.22266358, 0.22215056])
```

In [ ]:
```
# Work out explained proportion
df1_eigval["Var Explained"] = df_eigval["Eigenvalues"] / np.sum(df_eigval
df1_eigval = df_eigval[:5]

#Format as percentage
df1_eigval.style.format({"Var Explained": "{:.2%}"})
```

Out[ ]:

	Eigenvalues	Var Explained
0	19.753684	98.59%
1	0.263619	1.32%
2	0.016647	0.08%
3	0.002097	0.01%
4	0.000361	0.00%

In [ ]:
```
(df1_eigval['Var Explained'][:5]*100).iplot(kind='bar',
                                             title='Percentage Variance E
                                             color='cornflowerblue')
```

Visualize PCs

In [ ]:
```
# Subsume first 3 components into a dataframe
pcdf = pd.DataFrame(eigenvectors[:,0:3], columns=['e1','e2','e3'], index=
pcdf[:10]
```

Out[ ]:

	e1	e2	e3
1.0	0.218112	0.463225	-0.551906
1.5	0.220712	0.380888	-0.272863
2.0	0.222348	0.307708	-0.059870
2.5	0.223377	0.244910	0.088958
3.0	0.224043	0.190317	0.185746
3.5	0.224486	0.141776	0.241599
4.0	0.224782	0.097657	0.265486
4.5	0.224972	0.056822	0.264741
5.0	0.225074	0.018513	0.245486
5.5	0.225099	-0.017770	0.212737
6.0	0.225053	-0.052343	0.170475
6.5	0.224942	-0.085405	0.121843
7.0	0.224770	-0.117103	0.069317
7.5	0.224542	-0.147554	0.014816
8.0	0.224261	-0.176856	-0.040212
8.5	0.223931	-0.205097	-0.094708
9.0	0.223554	-0.232354	-0.147926
9.5	0.223131	-0.258698	-0.199362
10.0	0.222664	-0.284186	-0.248703

In [ ]:

```
pcdf.iplot(title='Principal Components for Gilts Curve (unscaled)', secon
```

Singular Value Decomposition using Sklearn PCA

In [ ]:
```
# Wrap two main stages into a pipeline
pipe = Pipeline([("scaler", StandardScaler()), ("pca", PCA())])
pipe.fit(df)
```

Out[ ]:
```
▸ Pipeline
▸ StandardScaler
▸ PCA
```

In [ ]:
```
# eigenvectors
pipe['pca'].components_[0]
```

Out[ ]:
```
array([0.218112  , 0.22071219, 0.22234786, 0.22337726, 0.22404304,
       0.22448583, 0.22478226, 0.2249718 , 0.225074  , 0.22509903,
       0.22505332, 0.22494209, 0.22477012, 0.22454186, 0.22426118,
       0.2239312 , 0.22355415, 0.22313142, 0.22266358, 0.22215056])
```

In [ ]:

```
# eigenvalues, reference here to eigenvalues being canonical variances
pipe['pca'].explained_variance_
```

Out[ ]:

```
array([1.97536839e+01, 2.63618514e-01, 1.66472447e-02, 2.09709989e-03,
       3.61048910e-04, 1.98613387e-05, 2.05085861e-06, 1.57775294e-07,
       2.18653002e-08, 5.24252217e-09, 1.03925798e-09, 2.41742878e-10,
       6.16971007e-11, 1.56643511e-11, 6.87603089e-12, 2.16126409e-12,
       7.60197645e-13, 2.01508718e-13, 3.27096425e-14, 7.77492535e-15])
```

In [ ]:

```
# explained variance ratio is R^2 statistic, eg 98.89% for our PC1 below
pipe['pca'].explained_variance_ratio_
```

Out[ ]:

```
array([9.85888404e-01, 1.31569604e-02, 8.30848850e-04, 1.04664349e-04,
       1.80196228e-05, 9.91261358e-07, 1.02356489e-07, 7.87442147e-09,
       1.09127726e-09, 2.61649516e-10, 5.18684208e-11, 1.20651673e-11,
       3.07924621e-12, 7.81793524e-13, 3.43176451e-13, 1.07866726e-13,
       3.79407734e-14, 1.00571169e-14, 1.63250852e-15, 3.88039456e-16])
```

In [ ]:

```
df1_eigval = pd.DataFrame({'Eigenvalues': pipe['pca'].explained_variance_
                    'Var Explained': pipe['pca'].explained_variance_ratio
df1_eigval = df1_eigval[:5]

#Format as percentage
df1_eigval.style.format({"Var Explained": "{:.2%}"})
```

Out[ ]:

	Eigenvalues	Var Explained
0	19.753684	98.59%
1	0.263619	1.32%
2	0.016647	0.08%
3	0.002097	0.01%
4	0.000361	0.00%

Percentage of variance explained and eigenvalues obtained by Sklearn PCA match eigenvalues obtained by numpy linear algebra functionality. **QED**

PCA Projection

Dot product operation effectively applies the linear transformation represented by the eigenvectors **to each row** of our original data, providing a new representation of the data in the space defined by the principal components.

Take a single row of curves dataset (forward or spot) as a vector $\mathbf{f}$ with dimensions $(1, 20)$, which is 1 row x 20 columns.

The projection of $\mathbf{f}$ onto the principal components is computed with the eigenvectors matrix $\mathbf{V}$ wiht dimensions $(20, 3)$ -- eigenvectors are in columns. Matrix $\mathbf{V}$ will be in transposed position with regard to the data row $\mathbf{f}$.

$$\mathbf{f}_{\text{projected}} = \mathbf{f} \cdot \mathbf{V}$$

The dot product is calculated as follows:

$$\mathbf{f}_{\text{projected}} = \sum_{j=1}^{20} f_j \cdot \mathbf{V}_{ji}$$

f_j is the j-th element (tenor) of curve row $\mathbf{f}$,

$\mathbf{V}_{ji}$ is the j-th component of the i-th eigenvector.

Resulting table is not the dataset of alternative curves! Its columns are projections, not evolution of rates at specific tenors.

In [ ]:
```
# Dot product below 'projects' principal components, onto the scaled data

df1_projections = df1.dot(eigenvectors)  # all 20 eigenvectors preserved

df1_projections.index = df.index
df1_projections.head(10)
```

Out[ ]:

years:	0	1	2	3	4	5	6	
1970-01-31	1.102175	0.509879	0.018864	0.055784	0.005776	0.005033	-0.000456	0.00
1970-02-28	0.776345	0.488407	-0.003268	0.039490	-0.004190	0.001963	-0.000825	0.00
1970-03-31	0.306236	0.330603	-0.015749	0.042838	-0.001795	0.002541	-0.000854	0.0C
1970-04-30	0.476186	0.291168	0.021121	0.109716	0.010959	0.002292	-0.001483	0.0C
1970-05-31	0.114939	0.418865	0.048168	0.153189	0.015166	0.003729	-0.001805	0.00
1970-06-30	-0.342257	0.429821	-0.096485	0.140175	0.007289	0.006368	-0.001054	0.00
1970-07-31	-0.395990	0.205768	-0.161776	0.065392	-0.002375	0.007756	0.000217	0.00
1970-08-31	-0.325605	0.121112	-0.122751	0.074666	-0.005811	0.004775	-0.001201	0.0C
1970-09-30	-0.374422	0.134884	-0.125019	0.059176	-0.008538	0.004050	-0.001180	0.0(
1970-10-31	-0.146490	0.211438	-0.140262	0.100464	0.001645	0.005772	-0.001345	0.0C

In [ ]:

```
#Check dimensions
df1_projections.shape
```

Out[ ]: (550, 20)

In [ ]:

```
# Plot all
df1_projections.iplot(title='Projections')

# data.T.iplot(title='Quasi curves') this plot not very useful, it will s
```

In [ ]:
```
df1_projections3 = df1.dot(eigenvectors[:, 0:3])  # only 3 eigenvectors a
df1_projections3.index = df.index

df1_projections3.shape
```

Out[ ]: (550, 3)

In [ ]:
```
df1_projections3.head(10)
```

Out[ ]:

years:	0	1	2
1970-01-31	1.102175	0.509879	0.018864
1970-02-28	0.776345	0.488407	-0.003268
1970-03-31	0.306236	0.330603	-0.015749
1970-04-30	0.476186	0.291168	0.021121
1970-05-31	0.114939	0.418865	0.048168
1970-06-30	-0.342257	0.429821	-0.096485
1970-07-31	-0.395990	0.205768	-0.161776
1970-08-31	-0.325605	0.121112	-0.122751
1970-09-30	-0.374422	0.134884	-0.125019
1970-10-31	-0.146490	0.211438	-0.140262

PC1: Curve Level via 10Y Yield

In [ ]:

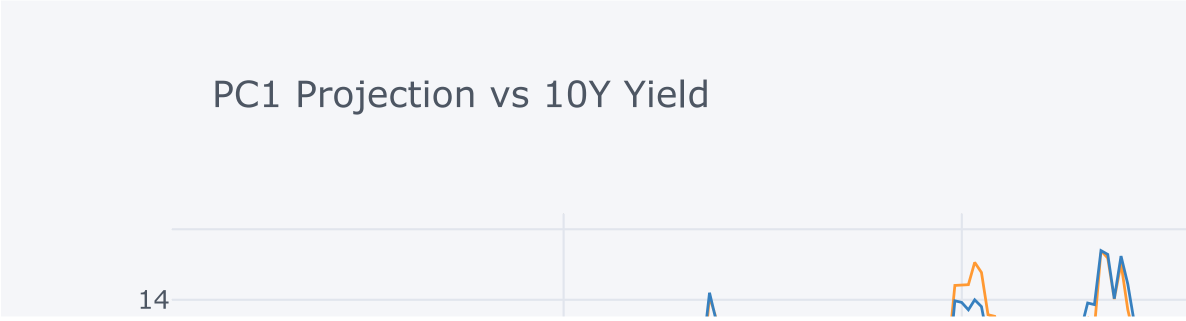
```
level = pd.DataFrame({'10Y': df[2.0],
                      'pc1_projection': df1_projections[0]})
level.head()
```

Out[ ]:

years:	10Y	pc1_projection
1970-01-31	8.700727	1.102175
1970-02-28	8.370748	0.776345
1970-03-31	7.795017	0.306236
1970-04-30	7.973522	0.476186
1970-05-31	7.862182	0.114939

In [ ]:

```
level.iplot(title='PC1 Projection vs 10Y Yield', secondary_y='pc1_project
```



PC2 : Slope

In [ ]:
```
# Calculate 10Y-2Y, typical measure of slope
slope = pd.DataFrame(df)
slope = slope[[2,10]]
slope['slope'] = slope[10] - slope[2]
slope['pc2_projection'] = - df1_projections[1] # here e2 demonstrated its
slope.head()
```

Out[ ]:

	2.0	10.0	slope	pc2_projection
years:				
1970-01-31	8.700727	8.279691	-0.421035	-0.509879
1970-02-28	8.370748	8.049074	-0.321674	-0.488407
1970-03-31	7.795017	7.845220	0.050204	-0.330603
1970-04-30	7.973522	8.041602	0.068079	-0.291168
1970-05-31	7.862182	7.630168	-0.232015	-0.418865

In [ ]:
```
slope[['slope', 'pc2_projection']].iplot(title='PC2 Projection vs 10Y-2Y
```

In [ ]:
```
# Verify the correlation
np.corrcoef(-df1_projections[1], slope['slope'])
```

Out[ ]:
```
array([[1.        , 0.98227356],
       [0.98227356, 1.        ]])
```

Correlation between the projection of PC2 and the slope of yield curve (10Y - 2Y) is near 1.

Confirms that the second principal component represents the slope type of movement.

References

[1] Bank of England YC Data.

[2] Scikit-learn PCA Decomposition.

[3] Richard Diamond (2014), PCA Application to Yield Curves note (distributed with HJM Lecture and this lab).

[4] UK Gilt Yields. Accessed December, 2023 but future work of this open source website not assured.

Addendum on Dot Product

In [ ]:
```
# Sample dataset (10 rows x 3 columns) representing interest rates (3 ten
rates_data = np.array([
    [0.038, 0.040, 0.045],
    [0.041, 0.042, 0.046],
    [0.044, 0.046, 0.048],
    [0.049, 0.048, 0.049],
    [0.046, 0.043, 0.047],
    [0.045, 0.044, 0.048],
    [0.047, 0.049, 0.046],
    [0.045, 0.047, 0.044],
    [0.039, 0.041, 0.050],
    [0.040, 0.043, 0.048]
])

# Example single eigenvector (1 x 3 dimensions)
eigenvector = np.array([[0.1, 0.2, 0.3]])

# Perform dot product
projected_data = rates_data.dot(eigenvector.T)

print("\nEigenvector:")
print(eigenvector)

print("\nProjected Data: alike to 10 daily values of 1 projection")
print(projected_data)
```

```
Eigenvector:
[[0.1 0.2 0.3]]

Projected Data: alike to 10 daily values of 1 projection
[[0.0253]
 [0.0263]
 [0.028 ]
 [0.0292]
 [0.0273]
 [0.0277]
 [0.0283]
 [0.0271]
 [0.0271]
 [0.027 ]]
```

In [ ]:

```python
# Now let's have 3 eigenvectors
eigenvectors = np.array([
    [0.1, 0.2, 0.3],
    [0.2, 0.3, 0.1],
    [0.3, 0.1, 0.2]
])

# Dot product with all 3 eigenvectors
projected_data = rates_data.dot(eigenvectors.T)

print("\nProjected Data: now we have 10 daily values of 3 projections")
print(projected_data)
```

```
Projected Data: now we have 10 daily values of 3 projections
[[0.0253 0.0241 0.0244]
 [0.0263 0.0254 0.0257]
 [0.028  0.0274 0.0274]
 [0.0292 0.0291 0.0293]
 [0.0273 0.0268 0.0275]
 [0.0277 0.027  0.0275]
 [0.0283 0.0287 0.0282]
 [0.0271 0.0275 0.027 ]
 [0.0271 0.0251 0.0258]
 [0.027  0.0257 0.0259]]
```