

initial_analysis

transforming dataset via principle component analysis¶

To create a parsimonious model for demonstrating the Bayesian technique, we will perform PCA on the raw dataset. This helps reduce noise and extract the most significant patterns of variation. Additionally, it requires fewer parameters than more complex affine term structure models.

Steps¶

- calculate log of change (don't use df.pcnt_ whatever it is)
- de mean the dataset
- calculate covariance matrix, eigenvalues and eigenvectors
- derive a calibration dataset
- attempt to fit normal or student-t distribution (student t better for heavier tails)
 - Q Q plot or histograms
 - q q plot great for seeing if normally distributed
- ?? BAYESIAN INFERENCE PARTS OF THE PROCESS ??
 - any hyperparameters
- ?? DERIVING STRESSES, COMPARING CLASSICAL VS BAYESIAN APPROACH ??

Deriving Stresses¶

$$Y_t = \log \left\{ \frac{X_t}{X_{t-1}} \right\}$$

simulate Y_t

$$\exp\{Y_t\} = \frac{X_t}{X_{t-1}}$$

$$X_{t-1} e^{Y_t} = X_t$$

X_{t-1} is current value of the curve and X_t value one year from now

the steps¶

- draw realisation of PC from probabilistic model
- (if using correlations) rescale using s.d. for each yield maturity
- add back the mean

¶

Imports¶

In [1]:

```
from IPython.display import display, Markdown
import numpy as np
import pandas as pd
from tabulate import tabulate
import matplotlib as plt
```

Getting Raw Data into Dataframes¶

We are initially putting the data into dataframes as these enable mixed data types (and we have here a mixture of dates and numerical values)

The bank of england provides two spreadsheets with historic spot yields at <https://www.bankofengland.co.uk/statistics/yield-curves/> we import each of these into a dataframes (df1 and df2) and join to make a single dataframe (df)

Import Libraries¶

Load in 1st spreadsheet¶

In [2]:

```
df1 = pd.read_excel("GLC Nominal month end data_1970 to 2015.xlsx",sheet_name="4. spot curves")
```

Headers¶

In [3]:

```
col_names=pd.read_excel("GLC Nominal month end data_1970 to 2015.xlsx",sheet_name="4. spot curves")
col_names[0]="Date"
df1.columns = col_names.iloc[0]
```



Checksum¶

spreadsheet¶ A manual highlight of cells in the spreadsheet

35	9.83	9.82	9.80	9.78	9.77	9.75	9.73
34	9.95	9.96	9.97	9.98	9.99	10.00	10.01
35	10.18	10.22	10.25	10.29	10.33	10.37	10.41
36	10.70	10.74	10.79	10.84	10.89	10.95	11.00
36	10.98	11.00	11.02	11.05	11.07	11.10	11.13
33	10.83	10.82	10.82	10.83	10.83	10.83	10.84

⋮

◀

Average: 7.55 Count: 25369 Sum: 191503.17  

shows the total value is 191503.17

numpy array In [4]:

```
df1.shape
```

Out[4]:

```
(552, 51)
```

The values run from row 0, column 1 to row 551, column 50 We put these values into a numpy array and calculate a sum that ignores nil values. Note np array references are not inclusive of value after the colon so we add 1 i.e. 0:552,1:52 and NOT 0:551,1:51.

In [5]:

```
print("The sum of values is from array is "+str(np.nansum(df1.iloc[0:552,1:51].to_numpy())))
```

```
The sum of values is from array is 191503.1723220289 .
```

and we see this is the same as sum of values from the spreadsheet.

Load in 2nd Spreadsheet

In [6]:

```
#load in second spreadsheet to df2
```

```
df2 = pd.read_excel("GLC Nominal month end data_2016 to present.xlsx",sheet_name="4. spot c
```

Headers

In [7]:

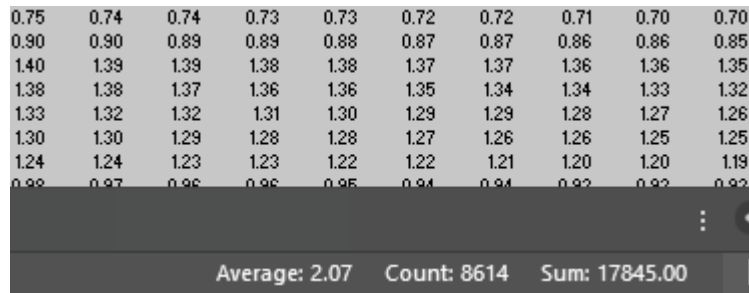
```
col_names2=pd.read_excel("GLC Nominal month end data_2016 to present.xlsx",sheet_name="4. sp
```

```
col_names2[0]="Date"
```

```
df2.columns = col_names2.iloc[0]
```

Checksum¶

spreadsheet¶ A manual highlight of cells in the spreadsheet



0.75	0.74	0.74	0.73	0.73	0.72	0.72	0.71	0.70	0.70
0.90	0.90	0.89	0.89	0.88	0.87	0.87	0.86	0.86	0.85
1.40	1.39	1.39	1.38	1.38	1.37	1.37	1.36	1.36	1.35
1.38	1.38	1.37	1.36	1.36	1.35	1.34	1.34	1.33	1.32
1.33	1.32	1.32	1.31	1.30	1.29	1.29	1.28	1.27	1.26
1.30	1.30	1.29	1.28	1.28	1.27	1.26	1.26	1.25	1.25
1.24	1.24	1.23	1.23	1.22	1.22	1.21	1.20	1.20	1.19
0.98	0.97	0.96	0.96	0.95	0.94	0.94	0.93	0.92	0.92
Average: 2.07 Count: 8614 Sum: 17845.00									

shows the total value is 17845.00

numpy array¶ In [8]:

```
df2.shape
```

```
Out[8]:
```

```
(108, 81)
```

The values run from row 0, column 1 to row 108, column 81 We put these values into a numpy array and calculate a sum that ignores nil values. Note np array references are not inclusive of value after the colon so we add 1 i.e. 0:109,1:82

```
In [9]:
```

```
print("The sum of values is from array is "+str(np.nansum(df2.iloc[0:109,1:82].to_numpy()))+)
```

```
The sum of values is from array is 17844.99933087668 .
```

and we see this is the same as sum of values from the spreadsheet.

Create Combined DataFrame¶

Problem of more columns¶

Joining 2 datasets¶

```
In [10]:
```

```
#join the two dataframes to create df
```

```
df = pd.concat([df1, df2], ignore_index=True)
```

```
print("The length of combined dataframe is "+str(len(df))+" rows")
```

```
The length of combined dataframe is 660 rows
```

Check Sum of Values¶

In [11]:

```
df.shape
```

Out[11]:

```
(660, 81)
```

In [12]:

```
print("The sum of values is from combined dataframe is "+str(np.nansum(df.iloc[0:661,1:108]))
The sum of values is from combined dataframe is 209348.17165290553 .
```

In [13]:

```
print("The sum of values is from dataframe 1 is "+str(np.nansum(df1.iloc[0:552,1:51]).to_numpy()))
print("The sum of values is from dataframe 2 is "+str(np.nansum(df2.iloc[0:109,1:82]).to_numpy()))
print("making a total of "+str(np.nansum(df1.iloc[0:552,1:51]).to_numpy()+np.nansum(df2.iloc[0:109,1:82]).to_numpy()))

The sum of values is from dataframe 1 is 191503.1723220289 .
The sum of values is from dataframe 2 is 17844.99933087668 .
making a total of 209348.17165290558
```

Check Size of Combined DataFrame¶

In [14]:

```
#producing some sense checks
display(Markdown("**Checking Dataframe 1 - 1970 to 2015**"))
print("the first dates is "+ str(df.iloc[0,0].strftime('%Y-%m-%d'))+" and the last is " +str(df.iloc[551,0].strftime('%Y-%m-%d')))
print("one would therefore expect 12 x 46yrs = 552 entries")
print("and indeed we see the number of rows in df is "+str(len(df1)))
```

Checking Dataframe 1 - 1970 to 2015

```
the first dates is 1970-01-31 and the last is 2015-12-31
one would therefore expect 12 x 46yrs = 552 entries
and indeed we see the number of rows in df is 552
```

In [15]:

```
display(Markdown("**Checking Dataframe 2 - 2015 to present**"))
print("the first dates is "+ str(df.iloc[552,0].strftime('%Y-%m-%d'))+" and the last is " +str(df.iloc[660,0].strftime('%Y-%m-%d')))
print("one would therefore expect 12 x 9yrs = 108 entries")
print("and indeed we see the number of rows in df is "+str(len(df2)))
```

Checking Dataframe 2 - 2015 to present

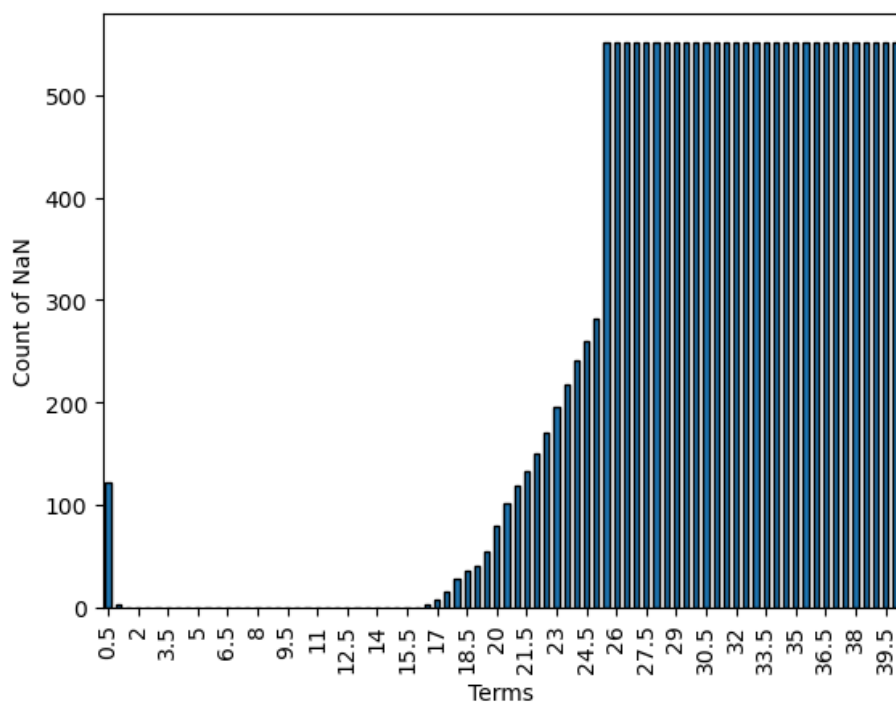
```
the first dates is 2016-01-31 and the last is 2024-12-31
one would therefore expect 12 x 9yrs = 108 entries
and indeed we see the number of rows in df is 108
```

Choosing Terms to Model¶

An inspection shows that there are a number of terms for which there is not a continuous set of data points:

In [16]:

```
nan_summary = df.iloc[0:661,1:108].isna().sum()
ax = nan_summary.plot.bar(edgecolor='black', xlabel="Terms", ylabel="Count of NaN")
ticks = ax.get_xticks();
ax.set_xticks(ticks[::3]); # Show every 5th label
```



we see that between terms 2 and 15 there are no missing values so we will choose to model this range for our analysis

NumPy Array with Terms of Interest¶

In [17]:

```
df.iloc[0:661,4:31].head()
```

Out[17]:

	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0	6.5
0	8.700727	8.664049	8.618702	8.572477	8.528372	8.487617	8.450611	8.417442	8.388098	8.360000
1	8.370748	8.337633	8.301590	8.265403	8.230804	8.198713	8.169617	8.143742	8.121153	8.100000
2	7.795017	7.793104	7.784963	7.775288	7.766459	7.759564	7.755068	7.753158	7.753877	7.750000
3	7.973522	8.002442	7.992813	7.967524	7.938335	7.911422	7.890054	7.875751	7.868985	7.860000
4	7.862182	7.877510	7.840673	7.782249	7.718053	7.656856	7.603548	7.560502	7.528577	7.500000

5 rows \times 27 columns

In [18]:

```
np_ToI = df.iloc[0:661,4:31].to_numpy()
```

Data Adjustments¶

Our aim is to model the log differences in spot yields. However there is a brief period around March 2020 where short term rates dipped below zero.

603	0.034725	0.033985	0.036396	0.040673	0.046292	0.053171	0.061445	0.071336	0.083063	0.096772	0.112521	0.130285	0.149969	0
604	-0.02646	-0.03836	-0.04537	-0.04828	-0.04748	-0.04317	-0.03547	-0.02451	-0.01042	0.006623	0.026389	0.048598	0.072939	0
605	-0.06968	-0.08247	-0.08781	-0.08777	-0.08313	-0.07464	-0.06257	-0.04722	-0.02884	-0.00772	0.015849	0.041565	0.069117	0
606	-0.09373	-0.11759	-0.13294	-0.14074	-0.1419	-0.1373	-0.12769	-0.11375	-0.09611	-0.0753	-0.05179	-0.02601	0.00168	0
607	-0.07268	-0.07709	-0.07402	-0.06418	-0.04849	-0.02785	-0.00305	0.025198	0.056286	0.089672	0.124869	0.161443	0.199012	0
608	-0.05385	-0.07152	-0.08223	-0.08549	-0.08174	-0.0717	-0.05616	-0.03591	-0.0117	0.015798	0.045965	0.078254	0.112178	0
609	-0.05319	-0.06333	-0.06733	-0.06497	-0.05668	-0.04305	-0.02473	-0.00235	0.023489	0.052251	0.083436	0.116576	0.151235	0
610	-0.02003	-0.02067	-0.01766	-0.01018	0.001872	0.018243	0.038508	0.06217	0.088734	0.117723	0.148696	0.181239	0.214973	0
611	-0.13525	-0.12803	-0.12028	-0.11067	-0.09832	-0.08284	-0.06419	-0.04257	-0.0183	0.008284	0.036825	0.066975	0.0984	0
612	-0.09564	-0.08745	-0.0757	-0.05969	-0.03939	-0.01516	0.012502	0.043034	0.075896	0.110587	0.146651	0.183678	0.221307	0
613	0.094282	0.138423	0.18449	0.23293	0.283586	0.33599	0.389559	0.443706	0.497903	0.551702	0.604718	0.656634	0.70719	0
614	0.072518	0.115257	0.163612	0.21638	0.27242	0.330658	0.390113	0.449941	0.509448	0.568086	0.625424	0.68112	0.734917	0
615	0.075907	0.119585	0.170169	0.225316	0.283404	0.34318	0.403628	0.463929	0.523452	0.581711	0.638328	0.693003	0.745504	0
616	0.058251	0.096404	0.142042	0.192965	0.247544	0.304464	0.36263	0.421167	0.479398	0.5368	0.592958	0.647531	0.700243	0
617	0.081577	0.123436	0.168623	0.215674	0.263858	0.312702	0.361837	0.41097	0.459864	0.508317	0.556141	0.603154	0.649178	0
618	0.071098	0.103739	0.138632	0.174753	0.21163	0.248999	0.286669	0.324484	0.362318	0.400056	0.437577	0.474746	0.511417	0
619	0.158168	0.185119	0.213648	0.244034	0.276237	0.310071	0.345269	0.381549	0.418635	0.456261	0.494165	0.53209	0.569787	0
620	0.331762	0.38646	0.436048	0.482952	0.528573	0.573684	0.618649	0.663583	0.708456	0.753146	0.797466	0.841195	0.884094	0
621	0.650299	0.682039	0.705514	0.726695	0.747964	0.77016	0.793459	0.817768	0.842893	0.868587	0.894568	0.920525	0.946138	0

These are problematic for the calculation of logs. For ease of analysis we remove these values and interpolate between the positive values in the month before and after the start of the -ve period

Removing Negatives¶

In [19]:

```
np_ToI_no_negs = np_ToI.copy()
np_ToI_no_negs[np_ToI_no_negs <= 0] = np.nan
```

600	0.461198	0.429297	0.408101	0.39386	0.384557	0.379173	0.37725	0.378627	0.383277	0.391206	0.402394	0.416767	0.434188	C
601	0.335304	0.313224	0.299199	0.290099	0.284427	0.281558	0.281322	0.283767	0.289024	0.297212	0.308399	0.322576	0.339662	C
602	0.139571	0.143826	0.146922	0.149716	0.152894	0.157051	0.162716	0.170375	0.180457	0.193285	0.209063	0.227861	0.249627	C
603	0.034725	0.033985	0.036396	0.040673	0.046292	0.053171	0.061445	0.071336	0.083063	0.096772	0.112521	0.130285	0.149969	C
604										0.006623	0.026389	0.048598	0.072939	C
605											0.015849	0.041565	0.069117	C
606													0.00168	C
607								0.025198	0.056286	0.089672	0.124869	0.161443	0.199012	C
608										0.015798	0.045965	0.078254	0.112178	C
609									0.023489	0.052251	0.083436	0.116576	0.151235	C
610					0.001872	0.018243	0.038508	0.06217	0.088734	0.117723	0.148696	0.181239	0.214973	C
611										0.008284	0.036825	0.066975	0.0984	C
612							0.012502	0.043034	0.075896	0.110587	0.146651	0.183678	0.221307	C
613	0.094282	0.138423	0.18449	0.23293	0.283586	0.33599	0.389559	0.443706	0.497903	0.551702	0.604718	0.656634	0.70719	C
614	0.072518	0.115257	0.163612	0.21638	0.27242	0.330658	0.390113	0.449941	0.509448	0.568086	0.625424	0.68112	0.734917	C
615	0.075907	0.119585	0.170169	0.225316	0.283404	0.34318	0.403628	0.463929	0.523452	0.581711	0.638328	0.693003	0.745504	C
616	0.058251	0.096404	0.142042	0.192965	0.247544	0.304464	0.36263	0.421167	0.479398	0.5368	0.592958	0.647531	0.700243	C
617	0.081577	0.123436	0.168623	0.215674	0.263858	0.312702	0.361837	0.41097	0.459864	0.508317	0.556141	0.603154	0.649178	C
618	0.071098	0.103739	0.138632	0.174753	0.21163	0.248999	0.286669	0.324484	0.362318	0.400056	0.437577	0.474746	0.511417	C
619	0.158168	0.185119	0.213648	0.244034	0.276237	0.310071	0.345269	0.381549	0.418635	0.456261	0.494165	0.53209	0.569787	C
620	0.331762	0.38646	0.436048	0.482952	0.528573	0.573684	0.618649	0.663583	0.708456	0.753146	0.797466	0.841195	0.884094	C
621	0.650299	0.682039	0.705514	0.726695	0.747964	0.77016	0.793459	0.817768	0.842893	0.868587	0.894568	0.920525	0.946138	C

Interpolating Gaps¶

In [20]:

```
# Convert to a DataFrame for interpolation
df_ToI_no_negs = pd.DataFrame(np_ToI_no_negs)

# Interpolate down the columns
df_ToI_no_negs_interpolated = df_ToI_no_negs.interpolate(method='linear', axis=0)

# Convert back to NumPy
np_ToI_no_negs_interpolated = df_ToI_no_negs_interpolated.to_numpy()
```

Log Yields¶

we calculate the natural log of the spot yields

In [21]:

```
np_ToI_logged = np.log(np_ToI_no_negs_interpolated)
```

603	0.034725	0.033985	0.036396	0.040673	0.046292	0.053171	0.061445	0.071336	0.083063	0.096772	0.112521	0.130285	0.149969	0.171411	C
604	0.040681	0.044429	0.051206	0.059899	0.039947	0.048182	0.058168	0.059801	0.076369	0.006623	0.026389	0.048598	0.072939	0.099068	C
605	0.046637	0.054873	0.066015	0.079125	0.033601	0.043192	0.054891	0.048267	0.069675	0.034306	0.015849	0.041565	0.069117	0.098202	C
606	0.052592	0.065317	0.080824	0.098351	0.027255	0.038202	0.051615	0.036732	0.062981	0.061989	0.070359	0.101504	0.00168	0.030934	C
607	0.058548	0.07576	0.095634	0.117576	0.020909	0.033212	0.048338	0.025198	0.056286	0.089672	0.124869	0.161443	0.199012	0.237246	C
608	0.064504	0.086204	0.110443	0.136802	0.014564	0.028222	0.045061	0.037522	0.039888	0.015798	0.045965	0.078254	0.112178	0.147299	C
609	0.070459	0.096648	0.125253	0.156028	0.008218	0.023233	0.041784	0.049846	0.023489	0.052251	0.083436	0.116576	0.151235	0.187014	C
610	0.076415	0.107092	0.140062	0.175253	0.001872	0.018243	0.038508	0.06217	0.088734	0.117723	0.148696	0.181239	0.214973	0.249557	C
611	0.082371	0.117535	0.154871	0.194479	0.095777	0.124159	0.025505	0.052602	0.082315	0.008284	0.036825	0.066975	0.0984	0.130781	C
612	0.088326	0.127979	0.169681	0.213705	0.189681	0.230075	0.012502	0.043034	0.075896	0.110587	0.146651	0.183678	0.221307	0.259217	C
613	0.094282	0.138423	0.18449	0.23293	0.283586	0.33599	0.389559	0.443706	0.497903	0.551702	0.604718	0.656634	0.70719	0.756191	C
614	0.072518	0.115257	0.163612	0.21638	0.27242	0.330658	0.390113	0.449941	0.509448	0.568086	0.625424	0.68112	0.734917	0.786627	C
615	0.075907	0.119585	0.170169	0.225316	0.283404	0.34318	0.403628	0.463929	0.523452	0.581711	0.638328	0.693003	0.745504	0.795658	C
616	0.058251	0.096404	0.142042	0.192965	0.247544	0.304464	0.36263	0.421167	0.479398	0.5368	0.592958	0.647531	0.700243	0.750875	C
617	0.081577	0.123436	0.168623	0.215674	0.263858	0.312702	0.361837	0.41097	0.459864	0.508317	0.556141	0.603154	0.649178	0.694046	C
618	0.071098	0.103739	0.138632	0.174753	0.21163	0.248999	0.286669	0.324484	0.362318	0.400056	0.437577	0.474746	0.511417	0.547436	C
619	0.158168	0.185119	0.213648	0.244034	0.276237	0.310071	0.345269	0.381549	0.418635	0.456261	0.494165	0.53209	0.569787	0.607019	C
620	0.331762	0.38646	0.436048	0.482952	0.528573	0.573684	0.618649	0.663583	0.708456	0.753146	0.797466	0.841195	0.884094	0.925928	C
621	0.650299	0.682039	0.705514	0.726695	0.747964	0.77016	0.793459	0.817768	0.842893	0.868587	0.894568	0.920525	0.946138	0.971088	C

Calculating Yield Differences¶

Spot Yield Differences¶

For each term of interest we calculate value of each natural log spot rate less the value of the natural log spot rate 1 year prior

In [22]:

```
lag = 12
diffs = np_ToI[lag:] - np_ToI[:-lag]
```

sense check¶

We are interested in terms 2 to 15. This includes half years so we have 27 columns. There are 660 rows of data. Since differences use a lag of 12 we will have 12 less rows 648.

In [23]:

```
print(np_ToI[0:648].shape)
print(np_ToI[12:661].shape)

(648, 27)
(648, 27)
```

The total value of differences should equate to the sum of rows 12:661 less sum of rows 0:648

In [24]:

```
np.sum(np_ToI[12:661]) - np.sum(np_ToI[0:648])
```

Out[24]:

```
np.float64(-1195.332302692288)
```

In [25]:

```
np.sum(diffs)
```

Out[25]:

```
np.float64(-1195.3323026922997)
```

spot check¶

we work through a couple of examples from raw data to the end to make sure same in output

De Mean¶

Calc Mean for Each Column and Deduct¶

In [26]:

```
# Compute the mean of each column
column_means = diffs.mean(axis=0)
print(column_means)
# Subtract the column means from the original array
demeaned_A = diffs - column_means

[-0.06410644 -0.06534015 -0.06597143 -0.06629206 -0.06644683 -0.06651528
 -0.06654538 -0.06656575 -0.06659277 -0.06663564 -0.06670102 -0.06679567
 -0.06692646 -0.06710001 -0.06732251 -0.06759952 -0.06793593 -0.06833595
 -0.06880317 -0.06934063 -0.06995092 -0.07063622 -0.07139829 -0.07223853
 -0.07315798 -0.0741573  -0.07523678]
```

Sense Checks¶

Check the means¶ In [27]:

```
diffs.shape
```

Out[27]:

```
(648, 27)
```

The sum of means * number of rows should aggregate to the sum of all entries

In [28]:

```
(column_means * 648).sum()
```

Out[28]:

```
np.float64(-1195.3323026922997)
```

In [29]:

```
diffs.sum()
```

Out[29]:

```
np.float64(-1195.3323026922997)
```

Check the de-meanned dataset¶ The sum of values in a demeaned dataset should equal zero. Furthermore, the sum of values in the original dataset less the mean * number of rows should also equal zero.

In [30]:

```
round(demeaned_A.sum(), 7)
```

Out[30]:

```
np.float64(-0.0)
```

In [31]:

```
print(diffs.sum() - (column_means * 648).sum())
```

0.0

spot check¶

we work through a couple of examples from raw data to the end to make sure same in output

Covariance or Correlation¶

In [32]:

```
type(demeaned_A)
```

Out[32]:

```
numpy.ndarray
```

In [33]:

```
covariance_matrix = np.cov(demeaned_A)
correlation_matrix = np.corrcoef(demeaned_A)
```

Eigenvectors and Eigenvalues¶

In [34]:

```
eigenvalues_from_covmatrix, eigenvectors_from_covmatrix = np.linalg.eig(covariance_matrix)
```

In [35]:

```
eigenvalues_from_correlmatrix, eigenvectors_from_correlmatrix = np.linalg.eig(correlation_matrix)
```



Incorporating Bayesian Framework¶

PCA reveals latent factors (

Having decided to model interest principle components, which economic outlooks correspond to these components:

Principle Component	Relevant Insights
PC1	level of interest rates - expected prolonged rates or gradual hiking against prolonged i
PC2	slope - short term vs long term expectations
PC3	curvature - short term vs long term expectations

could the bayesian rules enforce arbitrage freeness ??

- Economic theory imposes constraints of the first moments (see https://www.nber.org/system/files/working_papers/w24618/w24618.pdf)

Some links¶

<https://www.thegoldensource.com/pca-and-the-term-structure/#:~:text=The%20purpose%20of%20PCA%20is,14%20orthogonal%20lines%20using%20eigen vectors.>

In []: