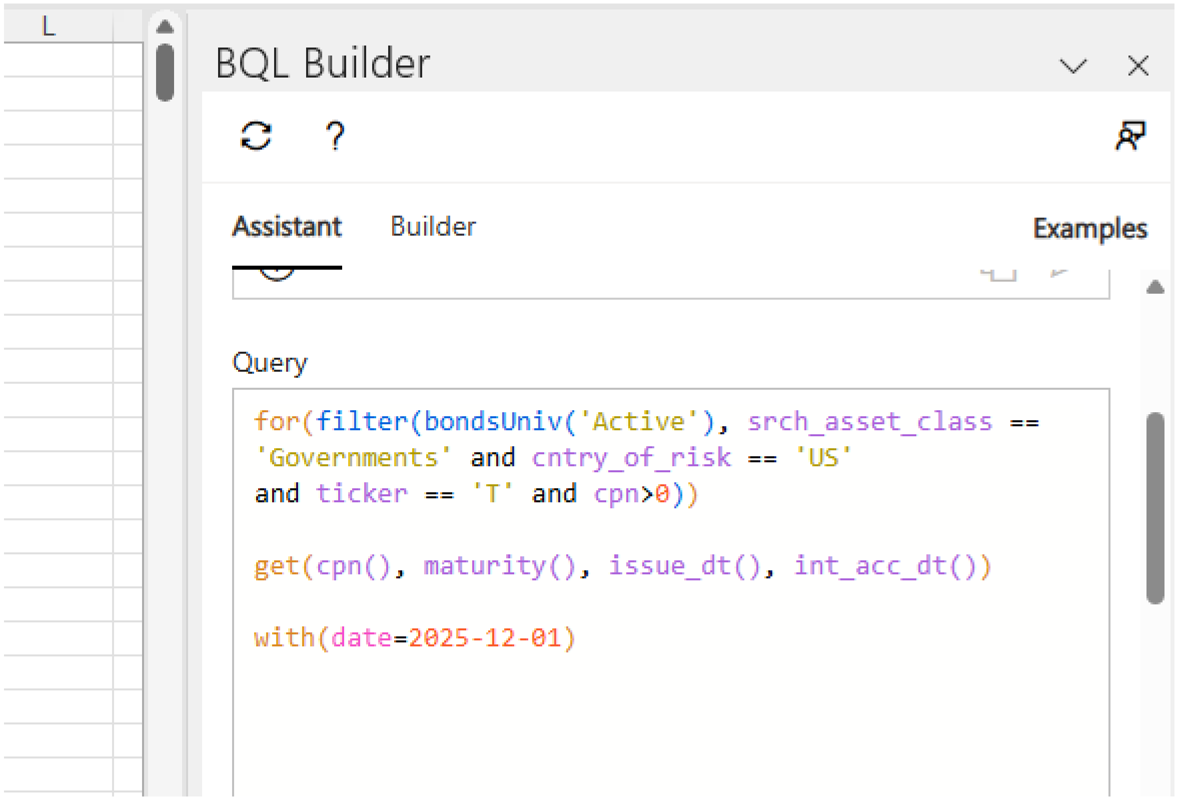
Interest Rate Derivatives, Homework #2 First Half, US Treasury Bonds

For the first half of this homework, we will be gathering US Treasury Bond data and writing python code to implement the Nelson-Siegel and Nelson-Siegel-Svensson splines. For the second half we will be using the spline for expected bond returns. The second half will be posted next week.

1) Create two data sets. One for the set of bonds that are active on December 1, 2025, which just gives indicative data (cusip, issue_date, dated_date, maturity_date, coupon). The second would just give price data history for all these bonds over the period from September 2, 2025 – December 1, 2025, a 3-month period. I suggest 3 columns: date, cusip, price. For the history, you can use BQL Builder in Excel. You should be logged into Bloomberg. Then in the Bloomberg tab at the top, select Bloomberg -> BQL Builder.
   In the Assistant tab, there is a Query section. Below is a screenshot that will get bond information for one date. You can simply change the single date to a range of dates for a history. Then you insert the query (press Insert button) and the data will appear in Excel.



```
for(filter(bondsUniv('Active'), srch_asset_class ==
'Governments' and cntry_of_risk == 'US'
and ticker == 'T' and cpn>0))

get(cpn(), maturity(), issue_dt(), int_acc_dt())

with(date=2025-12-01)
```

2) Create python code to implement a class for a Nelson-Siegel-Svensson bond spline and calibrator class. Helpful notes are below. If you get stuck, reach out for help.
3) Run the model over the data set for the NS model.
4) Run the model over the data set for the NSS model, all parameters free. Set bounds so that the taus can't overlap. For example tau1 between 2 and 10 years. Tau2 between 10 and 18 years.

5) Run the model over the data set for the NSS model, fixing the taus as:

$$\tau_1 = 4.0$$
$$\tau_2 = 10.75$$

6) Create a data history of parameters for 3), 4), 5). Create a parameter correlation matrix for each method and check if there is any difference between the methods. High correlations indicate factor interaction and typically cause large parameter changes day on day. Also look at the standard deviation of each parameter. A lower standard deviation indicates better stability of parameters. Compare the history of objective function size (result.fun from the optimizer) to see which method fits better. Compare the number of iterations of your solver to see if any model had problems converging.

Here is a suggestion for how you might structure the code:
class NelsonSiegelSvenssonSpline():
This class has data members for parameters and spline type (NS or NSS).
It has static methods for the function form of the zero coupon bond curves of both model types. For example,

```
@staticmethod
def nelsen_siegel_svenson_yield(maturity, params)
```

would just return the zcb yield given the maturity as a float. These are just the functional form / formuals for the spline.
The class would also have a few methods that fit the parameters to bond data (this one uses fixed taus):

```
def fit(self, bond_list, pvs_list, dv01_list, tau1, tau2, settlement_date):
```

You should be able to fit with or without fixed taus, and fit either NS or NSS.

The fit() functions would call an optimizer and have an associated objective function. Inside the fit function you have a call to an optimizer:

```
result1 = minimize(NelsenSiegelBondSpline.objective, x0=initial_params, args=args, method='L-BFGS-B', bounds = bounds, options = options)
```

The objective function calculates the error with the given parameters:

```
@staticmethod
def objective(params, *args):
…
total_error += (1/dv01_list[i])*(bond_pv_from_spline_list[i] – pv_from_market_list[i])**2
…
return total_error
```

You may find it useful to have a set up file that handles data input and data output.

class CalibrateNelsenSiegelBondSpline:

This class could be used to read in bond info (cusip, issue_date, dated_date, maturity_date, coupon) and create a list of bonds. Also it would read in a price history for all bonds. The class would hold all the data as well as a set of configurations for the spline. The would include fix_lambda (binary) and model (str: 'NS' or 'NSS'). The main function fit_spline() would instantiate the spline class and call the fit() function:

fit_spline(self, settlement_date, model='NSS', use_cubics_for_short_end=False, fix_lambdas=True):

The driver code in main() might look like this:

def main():

calibrator = CalibrateNelsenSiegelBondSpline() # reads in data and create price history, bond list

calibrator.configure_spline(model='NSS', fix_lambdas=True)

params, fun =calibrator.fit_spline(settlement_date, model='NSS', fix_lambdas=True)