RM 294 Optimization I Course Project

Dedicated portfolio of bonds using linear programming optimization techniques

Team Members:
John Flume, Ramya Madhuri Desineedi, Scott H Fields, Yue Cui

PROBLEM DESCRIPTION

Bonds are a form of debt issued by governments and companies. You can purchase a bond for a given price with a maturity date in the future at which point you will receive the face value of the bond. For example, you could purchase a bond for $98 (price) and at maturity the bond issuer will pay you $100 (face value). Coupon bonds are the bonds that offer a periodic payment during the life of the bond. Let’s say you purchase a bond for $102 with an annual coupon payment of $5, a face value of $100 and maturity three years from now. Then you will pay $102 immediately to purchase the bond, receive $5 coupon payments at the end of years 1, 2 and finally $5+$100=$105 at the end of year 3.

Bond forwards are similar to bonds, except the purchase of the bond will happen in the future, at terms agreed upon today. For example, if you enter a forward bond contract that starts at year 1 at a forward price of $98 with a maturity at year 4 and $3 coupon, then today no money exchanges hands, one year from today you must pay $98, two years from today you get a $3 payment, three years from today you get a $3 payment and 4 years from today you get a $103 payment. There is no optionality in this contract, you can choose not to enter the contract, but once you enter the contract you MUST pay the $98 next year.

Dedication or cash-flow matching is a technique used to fund known liabilities in the future. The intent is to form a portfolio of assets whose cash inflows will exactly match cash outflows. The liabilities will be paid off as they become due without the need to sell or buy assets in the future. The portfolio is formed today and held until all liabilities are paid off. Dedicated portfolios usually only consist of risk-free non-callable bonds and bond forwards since future cash inflows need to be known when the portfolio is constructed. Dedicated portfolios eliminate the risk caused by change in interest rates. Corporates, municipalities and pension funds routinely use such strategies. For example, corporates and municipalities sometimes want to fund liabilities stemming from projects that they might have initiated.

Our company is facing a stream of liabilities that it wants to be guaranteed to be able to pay using its cash reserves today. As team members in the office of CFO, you have decided to purchase a dedicated portfolio of bonds and forwards to hedge against interest rate risk.

APPROACH

Based on the liabilities our company has, we have formulated the portfolio construction problem as a linear programming problem.
Below are the steps we have followed which can be used to reconstruct on any general liabilities of the company and choice of bonds we have:

1. **Python Libraries:** Installing required libraries in python. We need 'Gurobi' for this.

```
## Importing necessary libraries
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import gurobipy as gp
import pandas as pd
from matplotlib import pyplot as plt
import matplotlib.pyplot as plt
```

2. **Reading liabilities and bonds data**

```
## read in data
liabilities = pd.read_csv('liabilities.csv')
bonds = pd.read_csv('bonds.csv')
```

3. **Getting number of variables to be solved (NumVar)**: This is number of bonds options we have in which we can invest money

```
## set number of variables
## in our case, all bond options are variables
NumVar = len(bonds)
NumVar
```

```
13
```

4. **Getting number of years of cashflow (n)**: This is obtained by taking maximum of 'Year' column from liabilities data

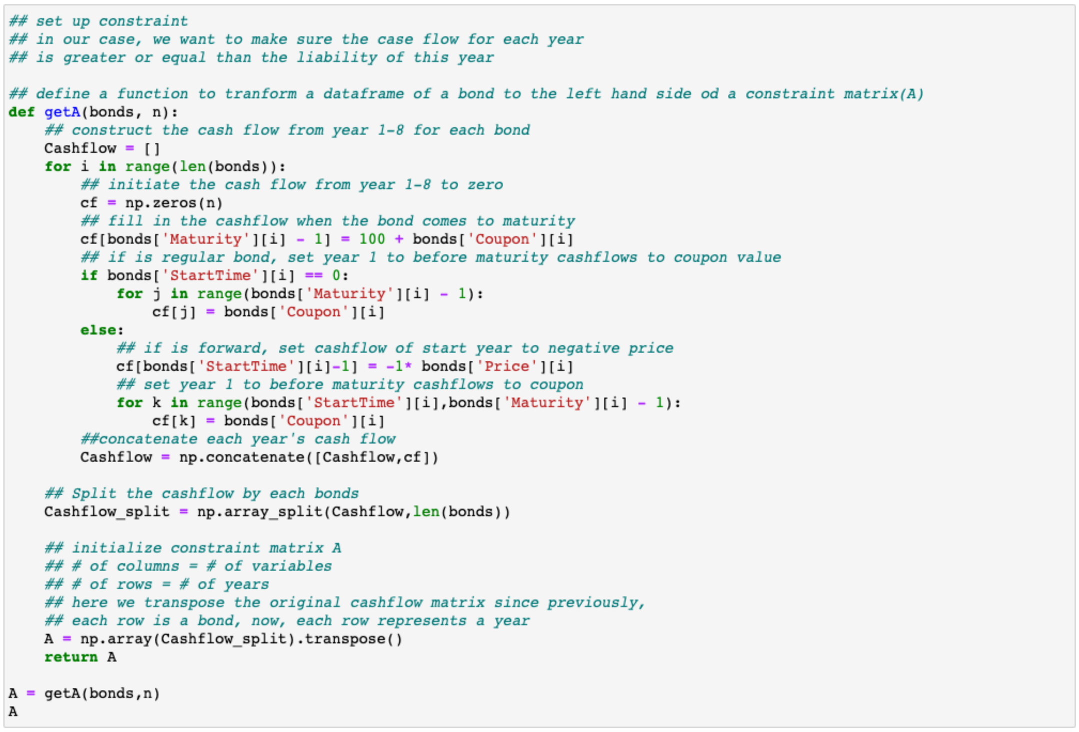
```
#Number of years for which we are designing cashflows
n = liabilities.Year.max()
n
```

```
8
```

5. **Setting up objective vectors (obj)** In this case, the object vectors are the cashflow we need to pay at year=0 for each bond/forward

```
## set up objective vectors
## the object vectors are the cashflow we need to pay at year=0 for each bond/forward
## define a function to tranform a dataframe of a bond to a list of cashflow
def getobj(bonds):
    obj = []
    ## loop through all bond option
    for i in range(len(bonds)):
        ## when year=0, pay price value for regular bond
        if bonds['StartTime'][i] == 0:
            obj.append(bonds['Price'][i])
        ## and 0 for forward
        else:
            obj.append(0)
    return obj

obj = getobj(bonds)
obj = np.array(obj)
obj
```

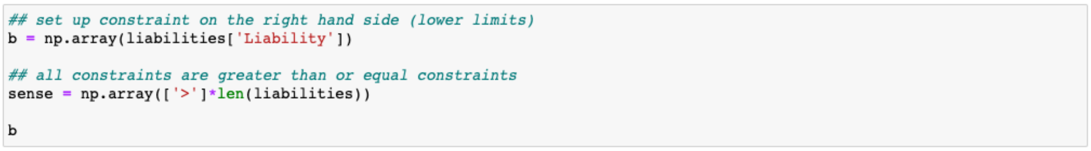
6. **Setting up constraint vectors (A):**
a. In our case, we want to make sure the case flow for each year is greater than or equal to the liability of that year
b. To do that, we have defined a function **"getA"** which takes as input - 'bonds' and 'n: number of years we are designing portfolio' (for example, if we have liabilities for 8 years and we are optimizing cash flows for these 8 years, then this number is 8)
c. Function getA checks whether it is a bond or bondforward and assigns cashflow (in and out) accordingly. The output is an array with cashflows for Year 1-n



```python
## set up constraint
## in our case, we want to make sure the case flow for each year
## is greater or equal than the liability of this year

## define a function to tranform a dataframe of a bond to the left hand side od a constraint matrix(A)
def getA(bonds, n):
    ## construct the cash flow from year 1-8 for each bond
    Cashflow = []
    for i in range(len(bonds)):
        ## initiate the cash flow from year 1-8 to zero
        cf = np.zeros(n)
        ## fill in the cashflow when the bond comes to maturity
        cf[bonds['Maturity'][i] - 1] = 100 + bonds['Coupon'][i]
        ## if is regular bond, set year 1 to before maturity cashflows to coupon value
        if bonds['StartTime'][i] == 0:
            for j in range(bonds['Maturity'][i] - 1):
                cf[j] = bonds['Coupon'][i]
        else:
            ## if is forward, set cashflow of start year to negative price
            cf[bonds['StartTime'][i]-1] = -1* bonds['Price'][i]
            ## set year 1 to before maturity cashflows to coupon
            for k in range(bonds['StartTime'][i],bonds['Maturity'][i] - 1):
                cf[k] = bonds['Coupon'][i]
        ##concatenate each year's cash flow
        Cashflow = np.concatenate([Cashflow,cf])

    ## Split the cashflow by each bonds
    Cashflow_split = np.array_split(Cashflow,len(bonds))

    ## initialize constraint matrix A
    ## # of columns = # of variables
    ## # of rows = # of years
    ## here we transpose the original cashflow matrix since previously,
    ## each row is a bond, now, each row represents a year
    A = np.array(Cashflow_split).transpose()
    return A

A = getA(bonds,n)
A
```

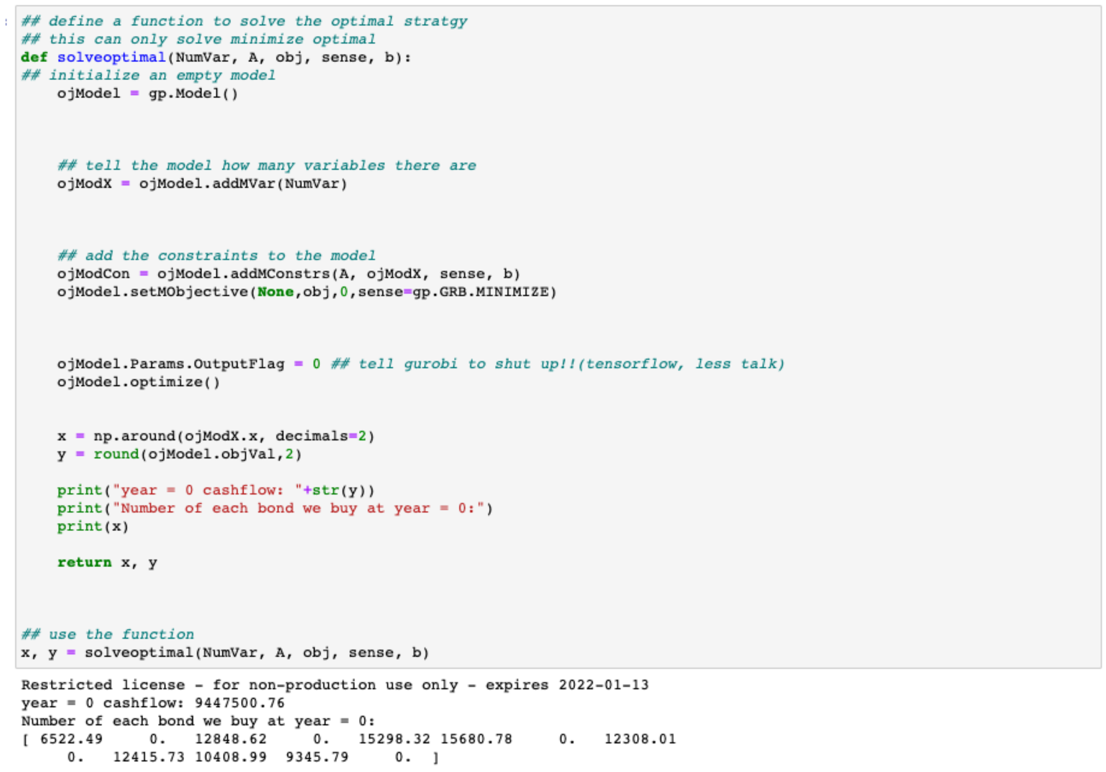
7. Setting up limits based on liabilities (b): This takes values from liabilities file

8. Setting up sense: As we are constructing portfolio to have net cashflows to be greater than liabilities, sense takes '>' value for all variables



```python
## set up constraint on the right hand side (lower limits)
b = np.array(liabilities['Liability'])

## all constraints are greater than or equal constraints
sense = np.array(['>']*len(liabilities))

b
```

9. **Optimal Strategy:** We have defined a function 'solveoptimal' which takes 'Numvar', 'A', 'obj', 'sense', 'b' which uses Gurobi to solve the problem using linear programming. It returns optimal values for all the variables we are trying to optimize. Below is the code snippet to optimize and corresponding results we got using the solver.

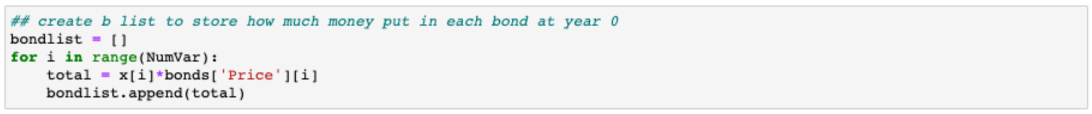


```
## define a function to solve the optimal stratgy
## this can only solve minimize optimal
def solveoptimal(NumVar, A, obj, sense, b):
## initialize an empty model
    ojModel = gp.Model()


    ## tell the model how many variables there are
    ojModX = ojModel.addMVar(NumVar)


    ## add the constraints to the model
    ojModCon = ojModel.addMConstrs(A, ojModX, sense, b)
    ojModel.setMObjective(None,obj,0,sense=gp.GRB.MINIMIZE)


    ojModel.Params.OutputFlag = 0 ## tell gurobi to shut up!!(tensorflow, less talk)
    ojModel.optimize()


    x = np.around(ojModX.x, decimals=2)
    y = round(ojModel.objVal,2)

    print("year = 0 cashflow: "+str(y))
    print("Number of each bond we buy at year = 0:")
    print(x)

    return x, y


## use the function
x, y = solveoptimal(NumVar, A, obj, sense, b)
```

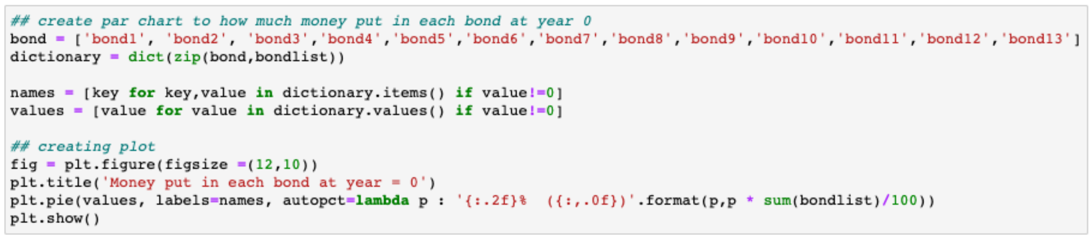
```
Restricted license - for non-production use only - expires 2022-01-13
year = 0 cashflow: 9447500.76
Number of each bond we buy at year = 0:
[ 6522.49     0.    12848.62     0.    15298.32 15680.78     0.    12308.01
     0.    12415.73 10408.99  9345.79     0.  ]
```

From the above output we can see the optimal solution in year 0 is to buy 6,522.49 bonds of bond 1, 0 of bond 2, 12,848.62 of bonds 3, 0 of bond 4, 15,298.32 of bond 5, 15,680.78 of bond 6, 0 of bond 7, 12,308.01 of bond 8, 0 of bond 9, 12,415.73 of bond 10, 10,408.99 of bond 11, 9,345.79 of bond 12, and finally 0 bonds of bond 13. This brings the total amount of money spent on bonds at $9,447,500.76 for year 0.

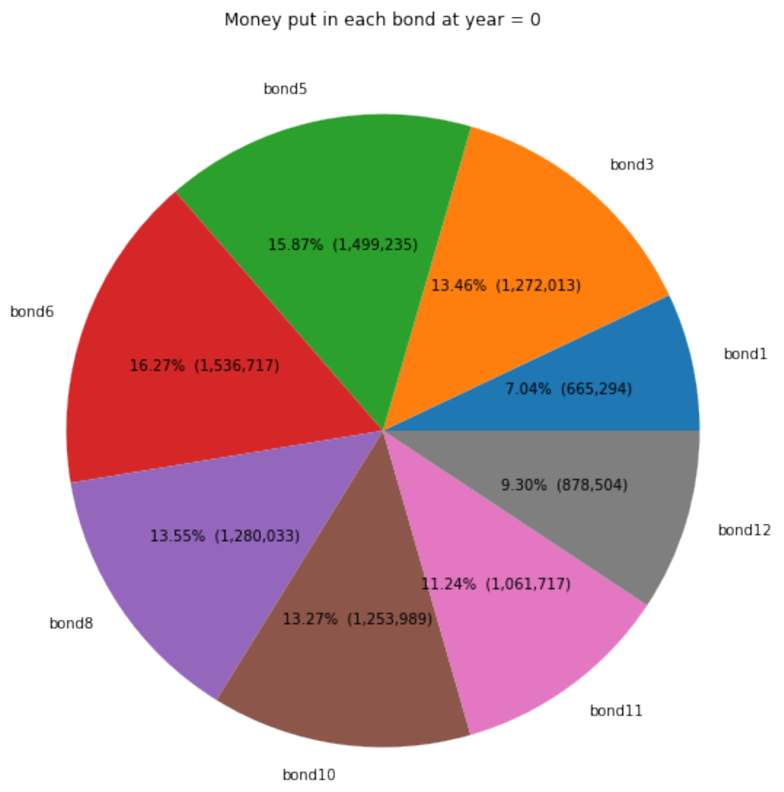
10. Optimal Solution with % of US$ (and Amount) invested in each bond:



```
## create b list to store how much money put in each bond at year 0
bondlist = []
for i in range(NumVar):
    total = x[i]*bonds['Price'][i]
    bondlist.append(total)
```

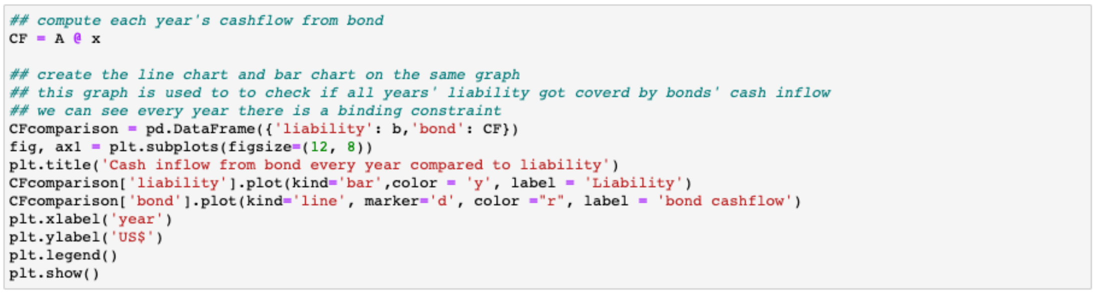


```
## create par chart to how much money put in each bond at year 0
bond = ['bond1', 'bond2', 'bond3','bond4','bond5','bond6','bond7','bond8','bond9','bond10','bond11','bond12','bond13']
dictionary = dict(zip(bond,bondlist))

names = [key for key,value in dictionary.items() if value!=0]
values = [value for value in dictionary.values() if value!=0]

## creating plot
fig = plt.figure(figsize =(12,10))
plt.title('Money put in each bond at year = 0')
plt.pie(values, labels=names, autopct=lambda p : '{:.2f}%  ({:,.0f})'.format(p,p * sum(bondlist)/100))
plt.show()
```

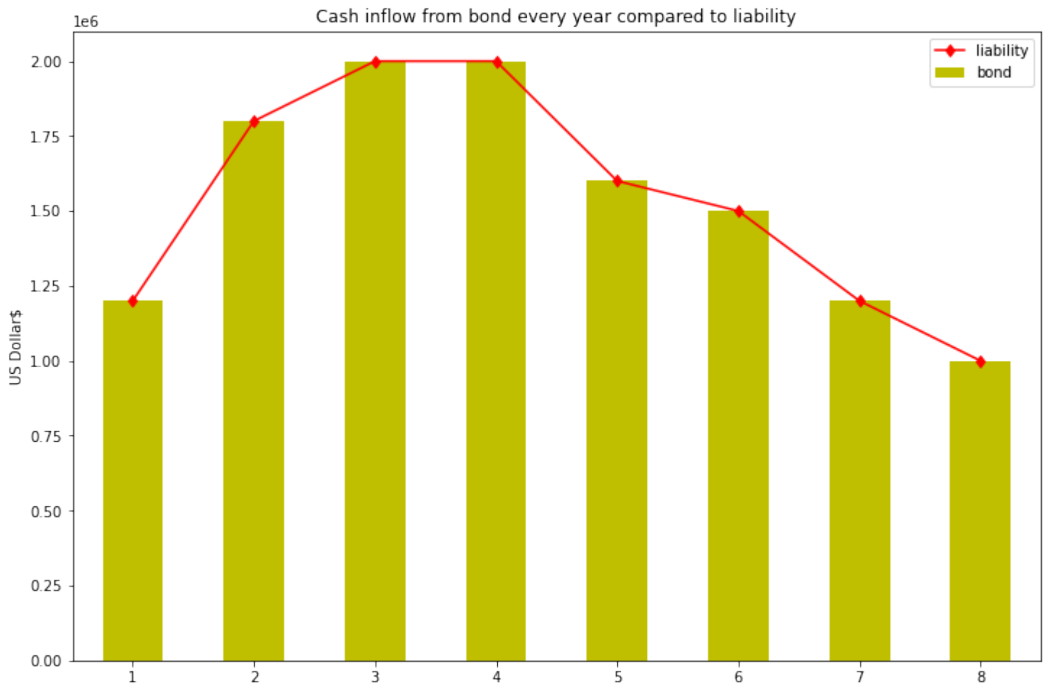


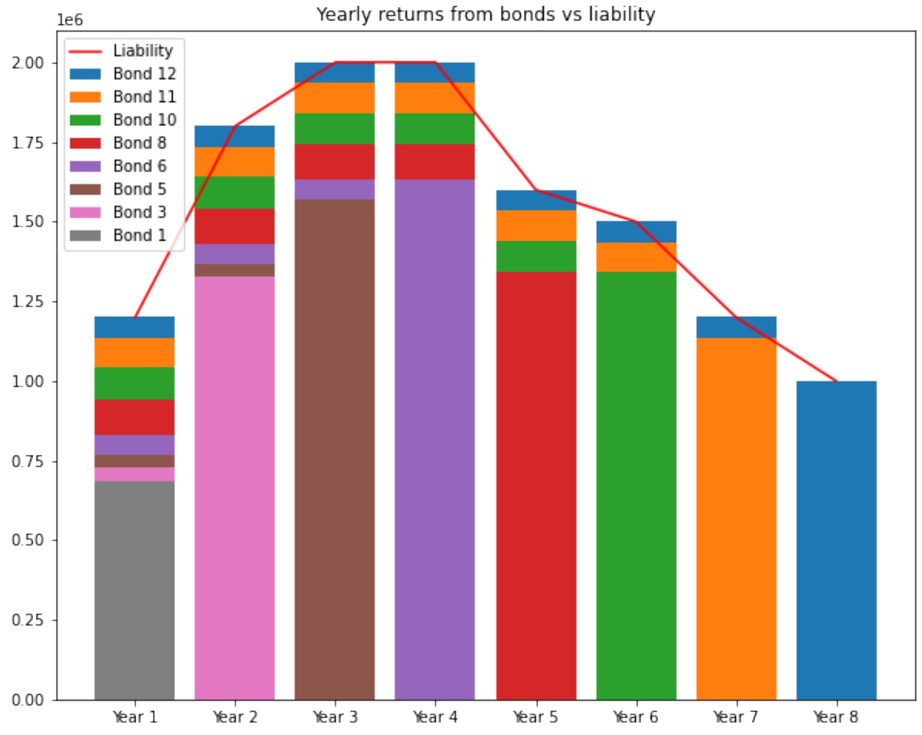
Following our optimal solution, the cash amount and proportion invested into each bond at year 0 can be seen in the above pie chart.

11. **Cashflows of our company with bonds and bond forwards:** The below graph shows the optimal cashflow from the bonds and liabilities of our company for each year. It demonstrates that our liabilities were indeed met by the pay outs from the purchased bonds. Below that shows the cash flow contributions from each bond.



```
## compute each year's cashflow from bond
CF = A @ x

## create the line chart and bar chart on the same graph
## this graph is used to to check if all years' liability got coverd by bonds' cash inflow
## we can see every year there is a binding constraint
CFcomparison = pd.DataFrame({'liability': b,'bond': CF})
fig, ax1 = plt.subplots(figsize=(12, 8))
plt.title('Cash inflow from bond every year compared to liability')
CFcomparison['liability'].plot(kind='bar',color = 'y', label = 'Liability')
CFcomparison['bond'].plot(kind='line', marker='d', color ="r", label = 'bond cashflow')
plt.xlabel('year')
plt.ylabel('US$')
plt.legend()
plt.show()
```





12. Sensitivity Analysis: Sensitivity analysis increases the confidence in the model and its predictions, by providing an under- standing of how the model responds to changes in the inputs. Adding a sensitivity analysis to a study means adding extra quality to it.

```
## since we only cares about the change on liability,
## we just need to focus on b, not c
ojModel = gp.Model()
ojModX = ojModel.addMVar(NumVar)
ojModCon = ojModel.addMConstrs(A, ojModX, sense, b)
ojModel.setMObjective(None,obj,0,sense=gp.GRB.MINIMIZE)
ojModel.Params.OutputFlag = 0 ## tell gurobi to shut up!!(tensorflow, less talk)
ojModel.optimize()


## find shadow price for each year's constraint
## every constraint seems to be binding
shadowprice = [con.Pi for con in ojModCon]

## lower bound for each year's liability(b) to keep the shadow price equation right
lowerbound = [round(con.SARHSLow,2) for con in ojModCon]

## upper bound for each year's liability(b) to keep the shadow price equation right
upperbound = [round(con.SARHSUp,2) for con in ojModCon]
```

```
## show the shadow price for each year's liability
shadow = pd.DataFrame({'liability(b)': b,'shadowprice': shadowprice,'lowerbound':lowerbound,'upperbound':upperbound})
shadow
```

	liability(b)	shadowprice	lowerbound	upperbound
0	1200000	0.971429	515138.37	inf
1	1800000	0.923671	470168.21	22052336.81
2	2000000	0.909876	431922.42	31062103.33
3	2000000	0.834424	369199.31	20890367.16
4	1600000	0.653628	258427.25	10751333.43
5	1500000	0.617183	159101.43	12618870.11
6	1200000	0.530350	65420.56	11972949.71
7	1000000	0.522580	-0.00	15820500.81

Our liabilities have a level of uncertainty due to the possibility of staff changes over the course of the portfolio. If employees leave the company, less liabilities will be needed, while the hiring of additional employees would create the need for additional liabilities. Because of this, the implementation of the above sensitivity analysis is helpful in understanding how changing the liabilities affects our objective. Since we only care about changes to our liabilities, we can ignore changes to other factors, such as our constraints. According to our sensitivity analysis, in the early years there is almost a 1 for 1 change in our liabilities to our objective of meeting those liabilities. This makes sense as the task at hand is matching our liabilities with a portfolio of cash flows. As time goes on, however, the shadow price becomes lower, meaning that a change of $1 of our liabilities changes our objective by a smaller fraction of $1 as time goes on. Eventually we see a shadow price of 0.52258, which is almost half of year 0. Thus, all else equal, our portfolio is less sensitive to later years. At each year we can also see that we have a very large range for the upper and lower bound and that our liabilities are well within the range, meaning we expect our shadow price to remain accurate. However, it is worth noting that the shadow price assumes we are only changing that year's liability.

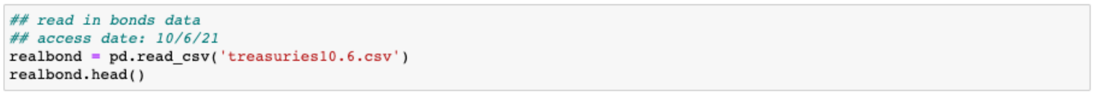
APPLYING TO REAL BOND MARKETS

Assumption:

- For simplicity, we assumed that all coupons are paid annually, even though in reality bond coupons are paid semi-annually.
- There are no forward bonds in this data set, hence we are ignoring forwards for this part.
- Assumption of now is 9/15/20

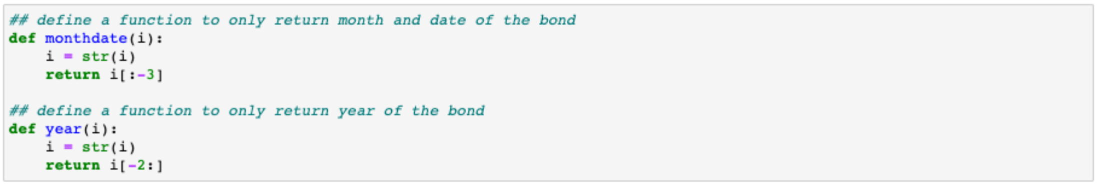
We accessed the bonds webpage on 6th October 2021. Along with the above assumptions we used the following steps to pre-process real bond data and then created a dedicated portfolio from real bonds to replicate the previous portfolio covering our liabilities:

1. Reading data downloaded from website

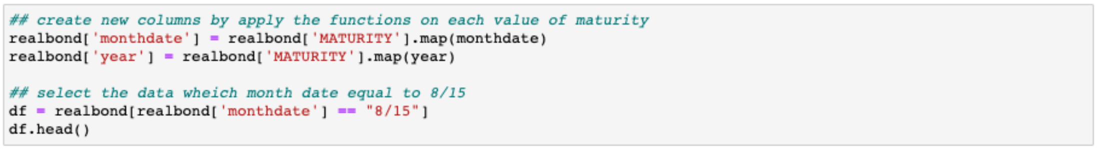


```
## read in bonds data
## access date: 10/6/21
realbond = pd.read_csv('treasuries10.6.csv')
realbond.head()
```

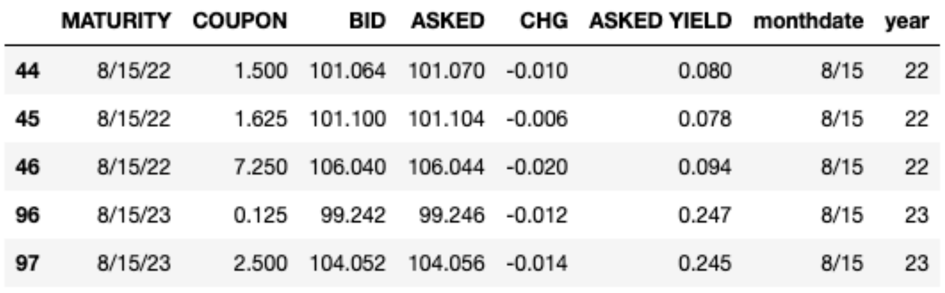
2. Defining functions that returns values of 'month date' and 'years for maturity of the bond' from 'maturity date' column



```
## define a function to only return month and date of the bond
def monthdate(i):
    i = str(i)
    return i[:-3]

## define a function to only return year of the bond
def year(i):
    i = str(i)
    return i[-2:]
```

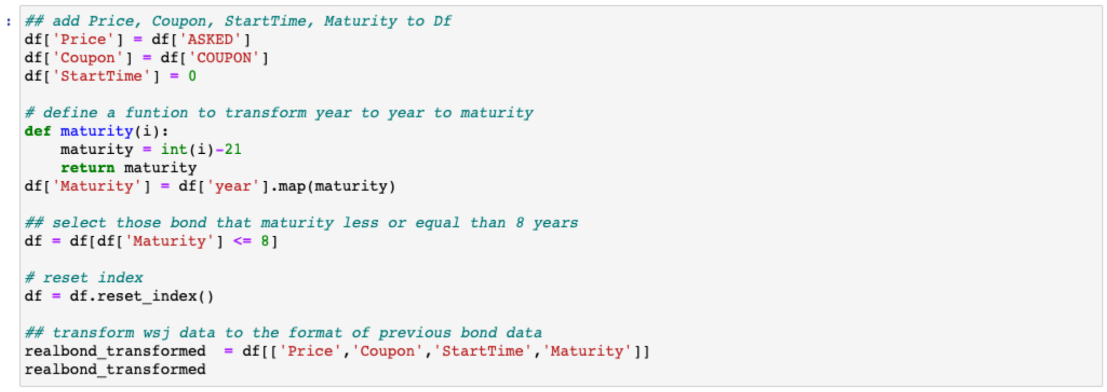


```
## create new columns by apply the functions on each value of maturity
realbond['monthdate'] = realbond['MATURITY'].map(monthdate)
realbond['year'] = realbond['MATURITY'].map(year)

## select the data wheich month date equal to 8/15
df = realbond[realbond['monthdate'] == "8/15"]
df.head()
```



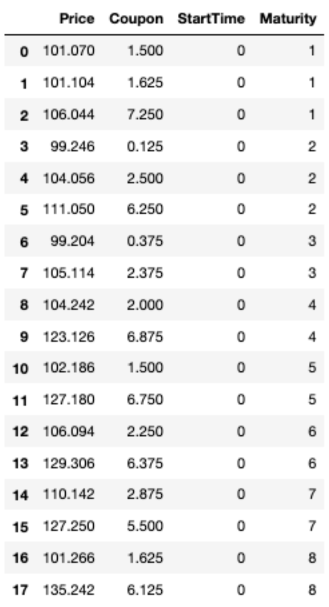
	MATURITY	COUPON	BID	ASKED	CHG	ASKED YIELD	monthdate	year
44	8/15/22	1.500	101.064	101.070	-0.010	0.080	8/15	22
45	8/15/22	1.625	101.100	101.104	-0.006	0.078	8/15	22
46	8/15/22	7.250	106.040	106.044	-0.020	0.094	8/15	22
96	8/15/23	0.125	99.242	99.246	-0.012	0.247	8/15	23
97	8/15/23	2.500	104.052	104.056	-0.014	0.245	8/15	23

3. **Pre-processing and transforming the data:** Other pre-processing steps we have done in python to transform data into the format which can be fed into the functions we have defined in the "Approach" section



```
## add Price, Coupon, StartTime, Maturity to Df
df['Price'] = df['ASKED']
df['Coupon'] = df['COUPON']
df['StartTime'] = 0

# define a funtion to transform year to year to maturity
def maturity(i):
    maturity = int(i)-21
    return maturity
df['Maturity'] = df['year'].map(maturity)

## select those bond that maturity less or equal than 8 years
df = df[df['Maturity'] <= 8]

# reset index
df = df.reset_index()

## transform wsj data to the format of previous bond data
realbond_transformed  = df[['Price','Coupon','StartTime','Maturity']]
realbond_transformed
```

4. **Reconstructed data frame:** We selected **18 bonds** for the time period that meet our needs, with up to 8 years maturity, which mature on 8/15. When we rearranged our columns into a data frame we used the listed ask price of the bonds as this is the current price that the seller is willing to sell the bond for, as seen below.



	Price	Coupon	StartTime	Maturity
0	101.070	1.500	0	1
1	101.104	1.625	0	1
2	106.044	7.250	0	1
3	99.246	0.125	0	2
4	104.056	2.500	0	2
5	111.050	6.250	0	2
6	99.204	0.375	0	3
7	105.114	2.375	0	3
8	104.242	2.000	0	4
9	123.126	6.875	0	4
10	102.186	1.500	0	5
11	127.180	6.750	0	5
12	106.094	2.250	0	6
13	129.306	6.375	0	6
14	110.142	2.875	0	7
15	127.250	5.500	0	7
16	101.266	1.625	0	8
17	135.242	6.125	0	8

5. **Constructing the optimal solution:** We have followed the steps from the "Approach" section to get the optimal solution for our investment in bonds.

```
## start solving optimal strategy
## set number of variables
## in our case, all bond options are variables
NumVar = len(realbond_transformed)
NumVar
```

```
18
```

```
## set up objective vectors
## the object vectors are the cashflow we need to pay at year=0 for each bond/forward
## there is no forwards ao all greater than zero
obj = getobj(realbond_transformed)
obj = np.array(obj)
obj
```

```
array([101.07 , 101.104, 106.044,  99.246, 104.056, 111.05 ,  99.204,
       105.114, 104.242, 123.126, 102.186, 127.18 , 106.094, 129.306,
       110.142, 127.25 , 101.266, 135.242])
```

```
## set up constraint
## in our case, we want to make sure the case flow for each year
## is greater or equal than the liability of this year
A = getA(realbond_transformed, n)

## check shape of constraint matrix
## # of columns = # of variables
## # of rows = # of years
A.shape
```

```
(8, 18)
```

```
## set up constraint on the right hand side (lower limits)
b = np.array(liabilities['Liability'])

## all constraints are greater than or equal constraints
sense = np.array(['>']*len(liabilities))
```

6. Output of Optimal Solution with Real Bonds:

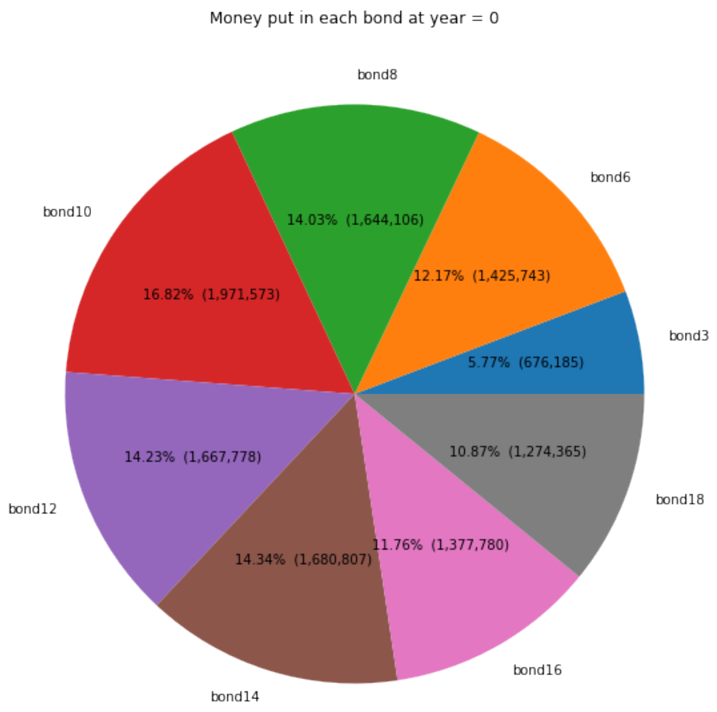
```
## find the optmal strategy for real bond
x,y = solveoptimal(NumVar, A, obj, sense, b)
```

```
year = 0 cashflow: 11718336.86
Number of each bond we buy at year = 0:
[    0.        0.     6376.46      0.        0.    12838.75      0.    15641.17
     0.    16012.65      0.    13113.52      0.    12998.68      0.    10827.35
     0.     9422.85]
```

As seen in the above code, the optimal portfolio consisted of 6376.46 bonds of bond 3, 12,838.75 of bond 6, 15,641.17 of bond 8, 16,012.65 of bond 10, 13,113.52 of bond 12, 12,998.68 of bond 14, 10,827.35 of bond 16, and 9,422.85 of bond 18. The rest of the bonds were not used. The total price paid for the bonds amounted to $11,718,336.86, which exceeds the previous portfolio.

7. Optimal Solution with % of US$ (and Amount) invested in each real bond:

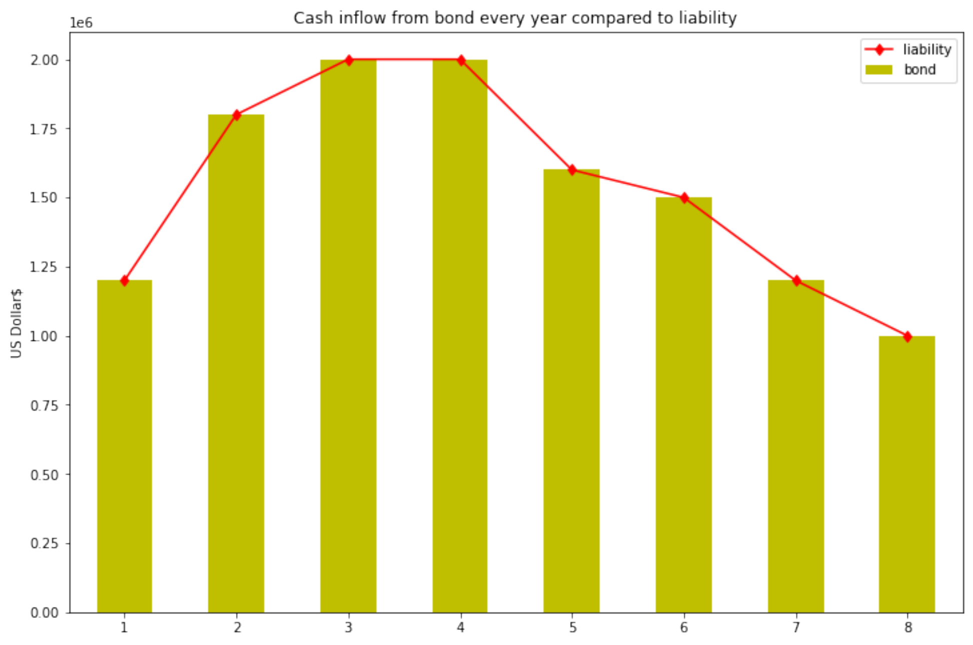
```
## create par chart to how much money put in each bond at year 0
bond = ['bond1', 'bond2', 'bond3','bond4','bond5','bond6','bond7','bond8','bond9','bond10','bond11','bond12','bond13','
dictionary = dict(zip(bond,bondlist))

names = [key for key,value in dictionary.items() if value!=0]
values = [value for value in dictionary.values() if value!=0]

## creating plot
fig = plt.figure(figsize =(12,10))
plt.title('Money put in each bond at year = 0')
plt.pie(values, labels=names, autopct=lambda p : '{:.2f}%  ({:,.0f})'.format(p,p * sum(bondlist)/100))
plt.show()
```

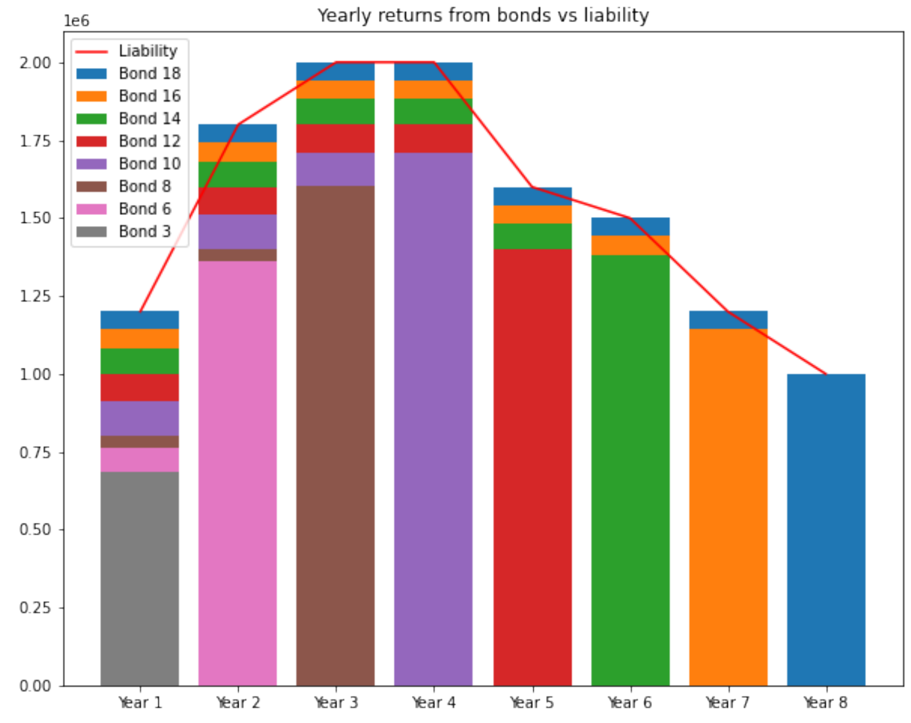


The proportion of actual cash invested into each bond can be seen again in the pie chart above.

8. Cashflows of our company with real bonds:

```
## compute each year's cashflow from bond
CF = A @ x

## create the line chart and bar chart on the same graph
## this graph is used to to check if all years' liability got coverd by bonds' cash inflow
## we can see every year there is a binding constraint
CFcomparison = pd.DataFrame({'liability': b,'bond': CF})
fig, ax1 = plt.subplots(figsize=(12, 8))
plt.title('Cash inflow from real bond portfolio every year compared to liability')
CFcomparison['liability'].plot(kind='bar',color = 'y', label = 'Liability')
CFcomparison['bond'].plot(kind='line', marker='d', color ="r", label = 'bond cashflow')
plt.xlabel('year')
plt.ylabel('US$')
plt.legend()
plt.show()
```





As with the original portfolio, we can see once again that our liabilities have been matched by the cash flows from the purchased bonds in the above graph. Thus, the replication of the portfolio using real bonds successfully matched the cash out flows of our liabilities.