In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
import math
import statsmodels.api as sm

from math import sqrt
from scipy.optimize import minimize
from scipy.stats import norm

import random
from array import array
```

Import Data

In [2]:
```
df= pd.read_csv('FRB_H15.csv')
df=df.drop(df.index[[0,1,2,3,4]])
df.columns = ['date', 1/12, 1/4, 1/2,1,2,3,5,7,10,20,30]

df=df[[1/12, 1/4, 1/2,1,2,3,5,7,10,20,30]]# drop column date
df=df.apply(pd.to_numeric)
df=df.reset_index(drop=True)


df.head()
```

Out[2]:

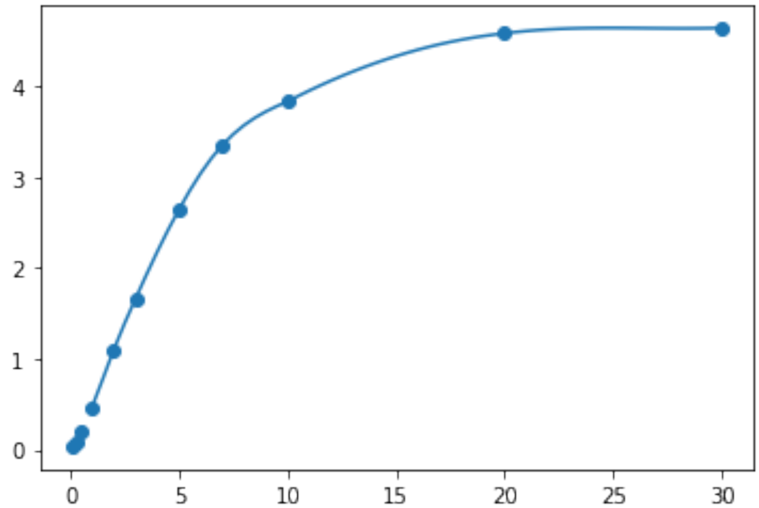
	0.08333333333333333	0.25	0.5	1	2	3	5	7	10	20	30
0	0.03	0.08	0.20	0.47	1.10	1.66	2.64	3.35	3.83	4.58	4.64
1	0.03	0.06	0.16	0.41	1.02	1.60	2.60	3.32	3.83	4.60	4.67
2	0.02	0.05	0.14	0.35	0.93	1.50	2.51	3.24	3.77	4.55	4.66
3	0.03	0.06	0.14	0.31	0.89	1.44	2.42	3.14	3.66	4.42	4.54
4	0.02	0.07	0.15	0.31	0.86	1.42	2.39	3.12	3.66	4.42	4.55

Installment 1

Interpolation

In [3]:
```
y = np.asfarray(df.iloc[-1].values.tolist())

def cubic_interp1d(y):
    """
    Interpolate a 1-D function using cubic splines.
      x0 : a float or an 1d-array
      x : (N,) array_like
          A 1-D array of real/complex values.
      y : (N,) array_like
          A 1-D array of real values. The length of y along the
          interpolation axis must be equal to the length of x.

    Implement a trick to generate at first step the cholesky matrice L
of
    the tridiagonal matrice A (thus L is a bidiagonal matrice that
    can be solved in two distinct loops).

    additional ref: www.math.uh.edu/~jingqiu/math4364/spline.pdf
    """
    x0= np.arange(1, 30.1, 0.5).tolist()
    x = np.asfarray([1/12, 1/4, 1/2,1,2,3,5,7,10,20,30])


    size = len(x)

    xdiff = np.diff(x)
    ydiff = np.diff(y)

    # allocate buffer matrices
    Li = np.empty(size)
    Li_1 = np.empty(size-1)
    z = np.empty(size)

    # fill diagonals Li and Li-1 and solve [L][y] = [B]
    Li[0] = sqrt(2*xdiff[0])
    Li_1[0] = 0.0
    B0 = 0.0 # natural boundary
    z[0] = B0 / Li[0]

    for i in range(1, size-1, 1):
        Li_1[i] = xdiff[i-1] / Li[i-1]
        Li[i] = sqrt(2*(xdiff[i-1]+xdiff[i]) - Li_1[i-1] * Li_1[i-1])
        Bi = 6*(ydiff[i]/xdiff[i] - ydiff[i-1]/xdiff[i-1])
        z[i] = (Bi - Li_1[i-1]*z[i-1])/Li[i]
```

```
    i = size - 1
    Li_1[i-1] = xdiff[-1] / Li[i-1]
    Li[i] = sqrt(2*xdiff[-1] - Li_1[i-1] * Li_1[i-1])
    Bi = 0.0 # natural boundary
    z[i] = (Bi - Li_1[i-1]*z[i-1])/Li[i]

    # solve [L.T][x] = [y]
    i = size-1
    z[i] = z[i] / Li[i]
    for i in range(size-2, -1, -1):
        z[i] = (z[i] - Li_1[i-1]*z[i+1])/Li[i]

    # find index
    index = x.searchsorted(x0)
    np.clip(index, 1, size-1, index)

    xi1, xi0 = x[index], x[index-1]
    yi1, yi0 = y[index], y[index-1]
    zi1, zi0 = z[index], z[index-1]
    hi1 = xi1 - xi0

    # calculate cubic
    f0 = zi0/(6*hi1)*(xi1-x0)**3 + \
         zi1/(6*hi1)*(x0-xi0)**3 + \
         (yi1/hi1 - zi1*hi1/6)*(x0-xi0) + \
         (yi0/hi1 - zi0*hi1/6)*(xi1-x0)
    return f0

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    x = np.asfarray([1/12, 1/4, 1/2,1,2,3,5,7,10,20,30])
    y = np.asfarray(df.iloc[0].values.tolist())
    plt.scatter(x, y)

    x_new = np.arange(1, 30.1, 0.5).tolist()
    plt.plot(x_new, cubic_interp1d(y))

    plt.show()
```



In [4]:
```
# cubic splines interpolated dataframe
i_df = np.array([])
for i in range(len(df)):
    y = np.asfarray(df.iloc[i].values.tolist())
    c0 = cubic_interp1d(y)
    i_df= np.append(i_df, c0, axis=0)

i_df=i_df/100
i_df.shape=(528,59)
df_itp=pd.DataFrame(i_df)
k= np.arange(1, 30.1, 0.5).tolist()
df_itp.columns = [k]
df_itp
```

Out[4]:

	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5
0	0.0047	0.007808	0.0110	0.013903	0.0166	0.019230	0.021763	0.024164	0.0264	0.02850
1	0.0041	0.007058	0.0102	0.013178	0.0160	0.018730	0.021320	0.023750	0.0260	0.02812
2	0.0035	0.006233	0.0093	0.012214	0.0150	0.017727	0.020340	0.022808	0.0251	0.02725
3	0.0031	0.005758	0.0089	0.011746	0.0144	0.017007	0.019535	0.021946	0.0242	0.02631
4	0.0031	0.005586	0.0086	0.011470	0.0142	0.016834	0.019335	0.021694	0.0239	0.02601
5	0.0033	0.005573	0.0083	0.011076	0.0138	0.016398	0.018835	0.021129	0.0233	0.02545
6	0.0035	0.005877	0.0086	0.011236	0.0138	0.016330	0.018788	0.021152	0.0234	0.02556
7	0.0036	0.006009	0.0089	0.011699	0.0144	0.017028	0.019545	0.021939	0.0242	0.02638
8	0.0034	0.005733	0.0086	0.011400	0.0141	0.016695	0.019157	0.021491	0.0237	0.02586
9	0.0034	0.005696	0.0084	0.011042	0.0136	0.016078	0.018452	0.020726	0.0229	0.02504
10	0.0039	0.006417	0.0093	0.012022	0.0146	0.017084	0.019463	0.021736	0.0239	0.02602

11	0.0041	0.006677	0.0097	0.012450	0.0150	0.017462	0.019827	0.022078	0.0242	0.02623
12	0.0042	0.007070	0.0105	0.013507	0.0162	0.018745	0.021150	0.023405	0.0255	0.02749
13	0.0043	0.007204	0.0106	0.013687	0.0165	0.019127	0.021577	0.023863	0.0260	0.02808
14	0.0047	0.007717	0.0111	0.014226	0.0171	0.019786	0.022274	0.024575	0.0267	0.02874
15	0.0044	0.007239	0.0105	0.013517	0.0163	0.018915	0.021349	0.023609	0.0257	0.02771
16	0.0044	0.007243	0.0105	0.013519	0.0163	0.018901	0.021311	0.023540	0.0256	0.02758
17	0.0043	0.007098	0.0103	0.013175	0.0158	0.018304	0.020680	0.022916	0.0250	0.02698
18	0.0039	0.006139	0.0088	0.011491	0.0141	0.016550	0.018814	0.020921	0.0229	0.02486
19	0.0038	0.005954	0.0085	0.011075	0.0136	0.016023	0.018308	0.020463	0.0225	0.02450
20	0.0035	0.005400	0.0078	0.010127	0.0124	0.014658	0.016859	0.018982	0.0210	0.02293
21	0.0036	0.005634	0.0081	0.010404	0.0126	0.014768	0.016876	0.018896	0.0208	0.02260
22	0.0036	0.005538	0.0079	0.010270	0.0126	0.014865	0.017023	0.019069	0.0210	0.02287
23	0.0033	0.005210	0.0075	0.009823	0.0121	0.014273	0.016319	0.018256	0.0201	0.02193
24	0.0030	0.005048	0.0075	0.009888	0.0122	0.014451	0.016607	0.018660	0.0206	0.02247
25	0.0029	0.004654	0.0069	0.009114	0.0113	0.013477	0.015601	0.017650	0.0196	0.02147
26	0.0031	0.004442	0.0062	0.008103	0.0101	0.012136	0.014148	0.016110	0.0180	0.01982
27	0.0031	0.004542	0.0063	0.008159	0.0101	0.012103	0.014108	0.016085	0.0180	0.01984
28	0.0028	0.004284	0.0063	0.008254	0.0102	0.012194	0.014200	0.016180	0.0181	0.01994
29	0.0027	0.004142	0.0060	0.007655	0.0093	0.011120	0.013081	0.015102	0.0171	0.01897
...	...	...	...	...	...	...	...	...	...	.
498	0.0195	0.018748	0.0182	0.017949	0.0179	0.017923	0.017994	0.018118	0.0183	0.01853
499	0.0198	0.018861	0.0184	0.018122	0.0180	0.017963	0.018003	0.018116	0.0183	0.01853
500	0.0194	0.018497	0.0181	0.017794	0.0176	0.017497	0.017485	0.017556	0.0177	0.01788
501	0.0178	0.016617	0.0161	0.015682	0.0154	0.015243	0.015204	0.015264	0.0154	0.01556
502	0.0177	0.016387	0.0156	0.015180	0.0150	0.014884	0.014802	0.014769	0.0148	0.01490
503	0.0175	0.016225	0.0154	0.014977	0.0148	0.014666	0.014554	0.014489	0.0145	0.01461
504	0.0175	0.016148	0.0152	0.014664	0.0144	0.014217	0.014081	0.014004	0.0140	0.01407
505	0.0172	0.015879	0.0150	0.014479	0.0142	0.013993	0.013844	0.013773	0.0138	0.01393
506	0.0181	0.017428	0.0169	0.016582	0.0164	0.016245	0.016127	0.016070	0.0161	0.01623
507	0.0186	0.017872	0.0173	0.016970	0.0168	0.016670	0.016579	0.016549	0.0166	0.01674
508	0.0179	0.017005	0.0165	0.016136	0.0159	0.015741	0.015656	0.015643	0.0157	0.01580
509	0.0166	0.015670	0.0149	0.014529	0.0144	0.014314	0.014251	0.014237	0.0143	0.01446

510	0.0163	0.015555	0.0150	0.014719	0.0146	0.014493	0.014402	0.014360	0.0144	0.01455
511	0.0160	0.015843	0.0159	0.015872	0.0158	0.015706	0.015642	0.015633	0.0157	0.01586
512	0.0159	0.015770	0.0160	0.016000	0.0159	0.015816	0.015805	0.015867	0.0160	0.01618
513	0.0157	0.015641	0.0159	0.015950	0.0159	0.015846	0.015841	0.015891	0.0160	0.01616
514	0.0158	0.016045	0.0164	0.016466	0.0164	0.016352	0.016388	0.016506	0.0167	0.01695
515	0.0156	0.015777	0.0162	0.016365	0.0164	0.016424	0.016497	0.016621	0.0168	0.01702
516	0.0155	0.015607	0.0159	0.015903	0.0158	0.015764	0.015839	0.015994	0.0162	0.01640
517	0.0159	0.015904	0.0161	0.016094	0.0160	0.015914	0.015898	0.015958	0.0161	0.01632
518	0.0157	0.015715	0.0158	0.015907	0.0160	0.016032	0.016049	0.016091	0.0162	0.01643
519	0.0155	0.015804	0.0163	0.016483	0.0165	0.016493	0.016536	0.016636	0.0168	0.01703
520	0.0153	0.015613	0.0163	0.016553	0.0166	0.016663	0.016812	0.017031	0.0173	0.01758
521	0.0153	0.015510	0.0162	0.016406	0.0164	0.016446	0.016617	0.016879	0.0172	0.01752
522	0.0157	0.015637	0.0157	0.015788	0.0159	0.016019	0.016168	0.016357	0.0166	0.01691
523	0.0154	0.015410	0.0156	0.015714	0.0158	0.015891	0.016018	0.016186	0.0164	0.01666
524	0.0154	0.015511	0.0158	0.015846	0.0158	0.015799	0.015892	0.016064	0.0163	0.01656
525	0.0155	0.015338	0.0152	0.015121	0.0151	0.015117	0.015185	0.015311	0.0155	0.01575
526	0.0150	0.014407	0.0141	0.013898	0.0138	0.013776	0.013822	0.013933	0.0141	0.01430
527	0.0149	0.014321	0.0141	0.014009	0.0140	0.013999	0.014017	0.014077	0.0142	0.01440

528 rows × 59 columns

In [64]:
```
df_itp.to_csv('cubic splines interpolated yield.csv', index=False)
```

Boostrapping to get spot rate

In [5]:
```
y= i_df[0]
t= np.arange(1, 30.1, 0.5).tolist()
def boostrap(y):
    s = [] # output array for spot rates
    for i in range(len(t)): #calculate i-th spot rate
        sum = 0
        for j in range(i): #by iterating through 0..i
            sum += y[i] / (1 + s[j])**t[j]
        value = ((1+y[i]) / (1-sum))**(1/t[i]) - 1
        s.append(value)
    return(s)
#boostrap(y)
```

In [6]:
```
# boostrapped dataframe
b_df = np.array([])
for i in range(len(i_df)):
    y = i_df[i]
    b0 = boostrap(y)
    b_df= np.append(b_df, b0, axis=0)

b_df.shape=(528,59)
df_bst=pd.DataFrame(b_df)
df_bst.columns = [k]

# select marketable maturities
frame1=df[[1/12,1/4,1/2]]
frame1=frame1.apply(lambda x: x/100)


df_bst=df_bst[[1,2,3,5,7,10,20,30]]

df_bst=frame1.reset_index(drop=True).join(df_bst)
df_bst.columns=[1/12, 1/4, 1/2,1,2,3,5,7,10,20,30]

df_bst
```

Out[6]:

	0.08333333333333333	0.25	0.5	1.0	2.0	3.0	5.0	7.0	10.
0	0.0003	0.0008	0.0020	0.0047	0.016616	0.028147	0.049768	0.067260	0.0
1	0.0003	0.0006	0.0016	0.0041	0.015403	0.027132	0.049037	0.066717	0.0
2	0.0002	0.0005	0.0014	0.0035	0.014039	0.025422	0.047330	0.065124	0.0
3	0.0003	0.0006	0.0014	0.0031	0.013434	0.024392	0.045568	0.062993	0.0
4	0.0002	0.0007	0.0015	0.0031	0.012978	0.024053	0.044980	0.062606	0.0
5	0.0004	0.0010	0.0017	0.0033	0.012520	0.023364	0.043803	0.062065	0.0
6	0.0005	0.0011	0.0018	0.0035	0.012974	0.023353	0.044003	0.062279	0.0

7	0.0006	0.0010	0.0019	0.0036	0.013430	0.024388	0.045563	0.063955	0.0
8	0.0008	0.0012	0.0019	0.0034	0.012975	0.023876	0.044582	0.062901	0.0
9	0.0009	0.0014	0.0019	0.0034	0.012671	0.023012	0.043011	0.061214	0.0
10	0.0011	0.0016	0.0022	0.0039	0.014035	0.024720	0.044917	0.063028	0.0
11	0.0013	0.0016	0.0024	0.0041	0.014642	0.025402	0.045474	0.062667	0.0
12	0.0012	0.0014	0.0024	0.0042	0.015861	0.027471	0.047980	0.064874	0.0
13	0.0014	0.0016	0.0024	0.0043	0.016012	0.027992	0.048969	0.066903	0.0
14	0.0016	0.0017	0.0025	0.0047	0.016770	0.029022	0.050320	0.067867	0.0
15	0.0015	0.0016	0.0024	0.0044	0.015858	0.027644	0.048373	0.065540	0.0
16	0.0014	0.0016	0.0024	0.0044	0.015858	0.027644	0.048166	0.065077	0.0
17	0.0014	0.0016	0.0024	0.0043	0.015554	0.026777	0.047003	0.063831	0.0
18	0.0011	0.0014	0.0022	0.0039	0.013275	0.023865	0.042939	0.059718	0.0
19	0.0015	0.0016	0.0022	0.0038	0.012819	0.023006	0.042189	0.059147	0.0
20	0.0017	0.0017	0.0023	0.0035	0.011759	0.020947	0.039299	0.055343	0.0
21	0.0016	0.0017	0.0023	0.0036	0.012213	0.021284	0.038870	0.053729	0.0
22	0.0013	0.0015	0.0022	0.0036	0.011909	0.021289	0.039274	0.054854	0.0
23	0.0008	0.0010	0.0018	0.0033	0.011304	0.020437	0.037525	0.052989	0.0
24	0.0004	0.0009	0.0016	0.0030	0.011307	0.020612	0.038521	0.054048	0.0
25	0.0006	0.0013	0.0019	0.0029	0.010397	0.019073	0.036613	0.052008	0.0
26	0.0013	0.0017	0.0022	0.0031	0.009335	0.017021	0.033541	0.048496	0.0
27	0.0017	0.0016	0.0020	0.0031	0.009486	0.017018	0.033541	0.048724	0.0
28	0.0016	0.0015	0.0020	0.0028	0.009489	0.017193	0.033735	0.048933	0.0
29	0.0015	0.0016	0.0020	0.0027	0.009035	0.015654	0.031838	0.047128	0.0
...	...	...	...	...	...	...	...	...	
498	0.0214	0.0211	0.0204	0.0195	0.027402	0.029984	0.033210	0.036556	0.0
499	0.0213	0.0209	0.0209	0.0198	0.027705	0.030148	0.033200	0.036548	0.0
500	0.0208	0.0208	0.0207	0.0194	0.027253	0.029470	0.032084	0.034979	0.0
501	0.0206	0.0203	0.0197	0.0178	0.024225	0.025756	0.027876	0.030407	0.0
502	0.0205	0.0195	0.0191	0.0177	0.023462	0.025084	0.026764	0.028852	0.0
503	0.0206	0.0196	0.0189	0.0175	0.023160	0.024748	0.026210	0.028484	0.0
504	0.0209	0.0199	0.0190	0.0175	0.022854	0.024066	0.025290	0.027128	0.0

505	0.0205	0.0197	0.0188	0.0172	0.022554	0.023731	0.024929	0.027359	0.0
506	0.0201	0.0196	0.0189	0.0181	0.025438	0.027449	0.029129	0.031715	0.0
507	0.0202	0.0195	0.0192	0.0186	0.026041	0.028122	0.030048	0.032666	0.0
508	0.0189	0.0188	0.0189	0.0179	0.024832	0.026603	0.028407	0.030554	0.0
509	0.0179	0.0178	0.0174	0.0166	0.022409	0.024082	0.025865	0.028526	0.0
510	0.0173	0.0170	0.0169	0.0163	0.022566	0.024425	0.026044	0.028714	0.0
511	0.0174	0.0166	0.0164	0.0160	0.023945	0.026467	0.028433	0.031188	0.0
512	0.0175	0.0166	0.0165	0.0159	0.024100	0.026638	0.029000	0.031770	0.0
513	0.0164	0.0159	0.0161	0.0157	0.023950	0.026643	0.029001	0.031566	0.0
514	0.0156	0.0155	0.0158	0.0158	0.024711	0.027488	0.030300	0.033525	0.0
515	0.0158	0.0158	0.0159	0.0156	0.024410	0.027498	0.030494	0.033517	0.0
516	0.0158	0.0157	0.0158	0.0155	0.023952	0.026472	0.029392	0.031961	0.0
517	0.0163	0.0161	0.0162	0.0159	0.024252	0.026806	0.029183	0.032164	0.0
518	0.0156	0.0156	0.0157	0.0157	0.023796	0.026819	0.029375	0.032568	0.0
519	0.0155	0.0156	0.0157	0.0155	0.024564	0.027668	0.030487	0.033511	0.0
520	0.0156	0.0157	0.0158	0.0153	0.024569	0.027844	0.031443	0.034698	0.0
521	0.0157	0.0158	0.0160	0.0153	0.024417	0.027502	0.031269	0.034720	0.0
522	0.0151	0.0155	0.0158	0.0157	0.023643	0.026650	0.030150	0.033773	0.0
523	0.0152	0.0154	0.0156	0.0154	0.023496	0.026483	0.029777	0.032974	0.0
524	0.0154	0.0156	0.0157	0.0154	0.023801	0.026476	0.029586	0.032777	0.0
525	0.0153	0.0155	0.0156	0.0155	0.022882	0.025287	0.028108	0.031247	0.0
526	0.0155	0.0156	0.0157	0.0150	0.021214	0.023088	0.025541	0.028176	0.0
527	0.0156	0.0157	0.0157	0.0149	0.021216	0.023433	0.025719	0.028567	0.0

528 rows × 11 columns

In [65]:
```
df_bst.to_csv('boostrap yield to spot rate.csv', index=False)
```

Installment 2

1. Interpolation of spot rates in monthly intervals

In [7]:
```
t_seq =k # 59 (# of columns)
zr_seq =b_df[-1] # 59 of columns

# use given tau, compute beta0, beta1, beta2, then output R(0,t)
def fitNSModel(tau, t_seq, zr_seq):
    t_to_tau = [ t/tau  for t in t_seq]
    xterm1 = [ (1.0-math.exp(-tt))/tt for tt in t_to_tau]
    xterm2 = [ (1.0-math.exp(-tt))/tt-math.exp(-tt) for tt in t_to_tau
]
    x = np.array([xterm1, xterm2]).T
    x = sm.add_constant(x)
    wt=np.append(t_seq[0],np.diff(t_seq))
    #Use the weighted OLS with the weight proportional to the tenor be
tween data points
    #This intends to give equal wt to the full yield curve rather than
overweight the portion with a lot of samples
    res = sm.WLS(zr_seq, x, wt).fit() # fit the best curve with given
tau

    params = res.params.tolist() # beta0, beta1, beta2
    x= x.tolist()# 1, number beside beta1, number beside beta2

    R_seq= [ np.dot(params, xi) for xi in x] # compute R(0,t) using fo
rmula in slide

    SSE=np.square(np.subtract(zr_seq,R_seq)).sum()# compute the sum of
squared errors between R(0,t) and r(0,t)

    return (SSE,params)

fitNSModel(10.099000000000009, t_seq, zr_seq)
```

Out[7]:
```
(3.3879736678722254e-05,
 [0.05787946935768776, -0.04141336798116013, -0.0013057856642409195]
)
```

In [8]:
```
# NOT given tau, find optimal tau and optimal betas and optimal x
def estNSParam(t_seq, zr_seq):
    #for yield curve estimation the search space in time is not likely
to be outside front part of the curve
    tau_univ = np.arange(0.1, 10.1, 0.01).tolist() # 100
    SSEs=[fitNSModel(tau, t_seq, zr_seq)[0] for tau in tau_univ] # 100

    opt_SSE= min(SSEs)
    opt_tau = tau_univ[np.argmin(SSEs)]
    opt_betas = fitNSModel(opt_tau, t_seq, zr_seq)[1]

    t_to_tau = [ t/opt_tau  for t in t_seq]
    xterm1 = [ (1.0-math.exp(-tt))/tt for tt in t_to_tau]
    xterm2 = [ (1.0-math.exp(-tt))/tt-math.exp(-tt) for tt in t_to_tau
]
    return (opt_SSE,opt_tau, opt_betas)

estNSParam(t_seq, zr_seq)
```

Out[8]:
```
(3.3883053104493914e-05,
 10.089999999999995,
 [0.057876534759534304, -0.041410906930359245, -0.001336337967324446
2])
```

In [9]:
```
# use optimal tau and betas to interpolate zero coupon in months for o
ne yield curve
# n is time step (1/12)
def NS_pred(zr_seq):
    predict_t_seq= np.arange(1/12, 30.01, 1/12)

    opt_Param= estNSParam(t_seq, zr_seq)
    opt_tau=opt_Param[1]
    opt_betas=opt_Param[2]

    #optimal xs:
    t_to_tau = [ t/opt_tau  for t in predict_t_seq]
    xterm1 = [ (1.0-math.exp(-tt))/tt for tt in t_to_tau]
    xterm2 = [ (1.0-math.exp(-tt))/tt-math.exp(-tt) for tt in t_to_tau
]
    x = np.array([xterm1, xterm2]).T
    x = sm.add_constant(x)
    x= x.tolist()# 1, number beside beta1, number beside beta2

    R_seq= [ np.dot(opt_betas, xi) for xi in x]

    return(R_seq)
NS_pred(zr_seq)
```

Out[9]:
```
[0.016630676288240012,
```

```
0.016794849202728918,
0.016958151790221136,
0.017120589235176238,
0.017282166689152903,
0.017442889271027093,
0.0176027620672065 4,
0.017761790131845 79,
0.017919978487058 81,
0.018077332123130 18,
0.018233855998725245,
0.018389555041098 39,
0.018544434146300 41,
0.018698498179384026,
0.018851751974608144,
0.019004200335640882,
0.019155848035761154,
0.019306699818058 64,
0.019456760395632784,
0.019606034451790256,
0.019754526640241014,
0.019902241585293114,
0.020049183882046337,
0.020195358096584 17,
0.020340768766164904,
0.020485420399411074,
0.020629317476497818,
0.020772464449339948,
0.020914865741777 68,
0.021056525749761 15,
0.021197448841533767,
0.021337639357814117,
0.021477101611976857,
0.021615839890232 28,
0.021753858451804706,
0.021891161529109544,
0.022027753327929417,
0.022163638027588 78,
0.022298819781127664,
0.022433302715473 99,
0.022567090931614908,
0.022700188504766867,
0.022832599484544 6,
0.022964327895128898,
0.023095377735433378,
0.023225752979269976,
0.023355457575513418,
0.023484495448264586,
0.023612870497012 7,
0.023740586596796537,
0.023867647598364402,
```

```
0.023994057328333193,
0.024119819589346256,
0.02424493816023024,
0.0243694167961 5089,
0.024493259228767796,
0.0246164691663 8807,
0.024739050294119047,
0.024861006274019826,
0.024982340745252043,
0.025103057324229298,
0.025223159604765878,
0.0253426511582 2426,
0.025461535533661796,
0.0255798162579 7626,
0.025697496836050517,
0.025814580750896172,
0.025931071463796283,
0.026046972414447086,
0.0261622870210 9877,
0.026277018680695343,
0.0263911707690 1352,
0.0265047466408 0068,
0.026617749629911914,
0.0267301830494 4613,
0.0268420501918 8129,
0.0269533543292 0868,
0.0270640987130 663,
0.027174286574871396,
0.0272839211259 5204,
0.027393005557677864,
0.027501543041589926,
0.0276095367295 2966,
0.0277169897537 6703,
0.027823905227127747,
0.0279302862431 197,
0.028036135876058483,
0.0281414571811 9212,
0.0282462531948 2495,
0.0283505269344 4066,
0.028454281398824527,
0.0285575195681 8479,
0.028660244404273255,
0.028762458850505098,
0.028864165832077793,
0.0289653682560 8935,
0.0290660690116 5569,
0.029166270970027212,
0.029265976984704706,
0.0293651898915 543,
0.029463912508921813,
```

```
0.029562147637746252,
0.02965989806167257,
0.029757166547163656,
0.0298539558436116,
0.029950268683448224,
0.0300461077822548,
0.030141475838871168,
0.030236375535503953,
0.030330809537834225,
0.03042478049512434,
0.030518291040324066,
0.03061134379017609,
0.03070394134532071,
0.030796086290399878,
0.03088778119416057,
0.030979028609557417,
0.031069831073854713,
0.03116019110872768,
0.031250111220363065,
0.03133959389955912,
0.03142864162182489,
0.031517256847478796,
0.03160544202174658,
0.03169319957485867,
0.031780531922146804,
0.031867441464140035,
0.03195393058666009,
0.03204000166091618,
0.032125657043599015,
0.032210899076974366,
0.03229573008897587,
0.03238015239329727,
0.0324641682894841,
0.03254778006302462,
0.03263098998544027,
0.03271380031437549,
0.03279621329368694,
0.032878231153532035,
0.03295985611045708,
0.03304109036748466,
0.03312193611420052,
0.03320239552683981,
0.03328247076837287,
0.03336216398859031,
0.0334414773241876,
0.03352041289884912,
0.03359897282333156,
0.03367715919554689,
0.03375497410064465,
0.03383241961109379,
```

```
0.033909497786763915,
0.03398621067500607,
0.034062560310732785,
0.034138548716497914,
0.034214177902575624,
0.0342894498670390 7,
0.034364366595838435,
0.034438930062878556,
0.034513142230095877,
0.034587005047535084,
0.0346605204534250 8,
0.0347336903742545 6,
0.034806516724847004,
0.0348790014084351 8,
0.0349511463167352 7,
0.0350229533300203 6,
0.035094424317193486,
0.035165561135860265,
0.0352363656324009 6,
0.035306839642042086,
0.035376984988927634,
0.035446803486189594,
0.0355162969360183 6,
0.0355854671297323 4,
0.0356543158478472 3,
0.0357228448601449 9,
0.035791055925742006,
0.0358589507931571 9,
0.0359265312003793 4,
0.0359937988749342 4,
0.0360607555339511 6,
0.0361274028842290 8,
0.0361937426223023 8,
0.036259776434506015,
0.0363255059970405 5,
0.0363909329760363 8,
0.0364560590276178 4,
0.0365208857979667 3,
0.0365854149233854 8,
0.0366496480303598 6,
0.036713586735621306,
0.0367772326462088 5,
0.036840587359530 6,
0.036903652463424835,
0.036966429536220716,
0.037028920146798525,
0.0370911258546496 5,
0.037153048209936014,
0.0372146887535492 2,
0.0372760490171692 2,
```

```
0.03733713052332276,
0.037397934785441206,
0.037458463307918174,
0.03751871758616676,
0.037578699106676244,
0.037638409347068656,
0.03769784977615474,
0.037757021853989756,
0.03781592703192872,
0.037874566752681464,
0.03793294245036715,
0.03799105555056864,
0.038048907470386285,
0.03810649961849154,
0.03816383339518009,
0.038220910192424795,
0.03827773139392806,
0.03833429837517408,
0.03839061250348064,
0.038446675138050494,
0.03850248763002263,
0.038558051322522977,
0.03861336755071491,
0.038668437641849324,
0.03872326291531454,
0.03877784468268569,
0.038832184247773925,
0.03888628290667523,
0.03894014194781893,
0.0389937626520159,
0.039047146292506454,
0.03910029413500787,
0.03915320743776169,
0.03920588745158065,
0.039258335419895325,
0.039310552578800444,
0.03936254015710096,
0.03941429937635773,
0.03946583145093301,
0.039517137588035545,
0.039568218987765386,
0.03961907684315852,
0.039669712340231075,
0.03972012665802327,
0.03977032096864314,
0.03982029643730991,
0.0398700542223971,
0.03991959547547538,
0.039968921341355095,
0.04001803295812858,
```

```
0.04006693145721209 5,
0.0401156179633876 4,
0.0401640935948443 2,
0.0402123594632195 9,
0.0402604166736401 3,
0.0403082663247625 3,
0.0403559095088136 6,
0.04040334731163075 5,
0.0404505808127013 5,
0.0404976110852028 2,
0.04054443919604172 6,
0.0405910662058929 6,
0.04063749316923849 5,
0.0406837211344060 4,
0.04072975114360731 5,
0.0407755842329761 6,
0.0408212214326063 7,
0.0408666637665892 6,
0.0409119122530510 5,
0.040956967904189 9,
0.0410018317263128 2,
0.0410465047198722 9,
0.0410909878795026 3,
0.0411352821940561 5,
0.0411793886466390 2,
0.0412233082146470 4,
0.0412670418698010 1,
0.04131059057818197 6,
0.0413539553002662 2,
0.04139713699096,
0.0414401365996341 6,
0.0414829550701583 1,
0.0415255933409350 8,
0.0415680523449338 4,
0.04161033300972446 4,
0.04165243625751071 5,
0.0416943630051634 3,
0.041736114164253 6,
0.04177769064108509 4,
0.0418190933367272 7,
0.041860323147047 3,
0.04190138096274239 5,
0.0419422676693716 7,
0.04198298414738796 4,
0.0420235312721693 3,
0.04206390991405037 4,
0.0421041209383533 9,
0.042144165205419 3,
0.0421840435706383 7,
0.04222375688448075 4,
```

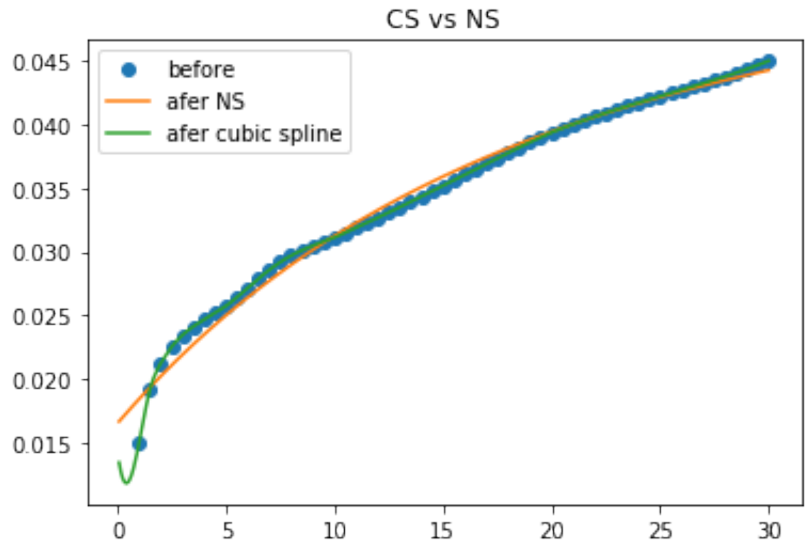
```
0.042263305992526805,
0.04230269173549724,
0.0423419149492831,
0.04238097646497541,
0.04241987710889487,
0.04245861770262112,
0.04249719906302197,
0.04253562200228239,
0.042573887327933276,
0.042611995842880106,
0.042649948345431385,
0.042687745629326854,
0.042725388483765586,
0.04276287769343388,
0.04280021403853295,
0.042837398294806484,
0.04287443123356797,
0.04291131362172789,
0.04294804622182072,
0.042984629792031756,
0.0430210650862238,
0.0430573528539636,
0.04309349384054819,
0.04312948878703103,
0.04316533843024801,
0.04320104350284323,
0.043236604733294666,
0.04327202284593964,
0.04330729856100015,
0.04334243259460802,
0.04337742565882991,
0.0434122784616921,
0.043446991707205236,
0.04348156609538881,
0.0435160023222955,
0.04355030108003543,
0.043584463056800166,
0.04361848893688664,
0.043652379400720885,
0.043686135124881634,
0.04371975678212372,
0.04375324504140144,
0.04378660056789161,
0.04381982402301663,
0.04385291606446727,
0.04388587734622543,
0.043918708518586626,
0.043951410228182476,
0.04398398311800289,
0.04401642782741824,
```

```
0.04404874499220137,
0.04408093524454936,
0.0441129992131053,
0.04414493752297982,
0.04417675079577254,
0.04420843964959334,
0.04424000469908352,
0.04427144655543684,
0.04430276582642038 4]
```

cubic spline vs NS interpolation on most recent spot rate curve

In [10]:
```
def cubic_interp1d(y):
    x0= np.arange(1/12, 30.01, 1/12).tolist()
    x = np.arange(1, 30.1, 0.5)


    size = len(x)

    xdiff = np.diff(x)
    ydiff = np.diff(y)

    # allocate buffer matrices
    Li = np.empty(size)
    Li_1 = np.empty(size-1)
    z = np.empty(size)

    # fill diagonals Li and Li-1 and solve [L][y] = [B]
    Li[0] = sqrt(2*xdiff[0])
    Li_1[0] = 0.0
    B0 = 0.0 # natural boundary
    z[0] = B0 / Li[0]

    for i in range(1, size-1, 1):
        Li_1[i] = xdiff[i-1] / Li[i-1]
        Li[i] = sqrt(2*(xdiff[i-1]+xdiff[i]) - Li_1[i-1] * Li_1[i-1])
        Bi = 6*(ydiff[i]/xdiff[i] - ydiff[i-1]/xdiff[i-1])
        z[i] = (Bi - Li_1[i-1]*z[i-1])/Li[i]

    i = size - 1
    Li_1[i-1] = xdiff[-1] / Li[i-1]
    Li[i] = sqrt(2*xdiff[-1] - Li_1[i-1] * Li_1[i-1])
    Bi = 0.0 # natural boundary
    z[i] = (Bi - Li_1[i-1]*z[i-1])/Li[i]
```

```
    # solve [L.T][x] = [y]
    i = size-1
    z[i] = z[i] / Li[i]
    for i in range(size-2, -1, -1):
        z[i] = (z[i] - Li_1[i-1]*z[i+1])/Li[i]

    # find index
    index = x.searchsorted(x0)
    np.clip(index, 1, size-1, index)

    xi1, xi0 = x[index], x[index-1]
    yi1, yi0 = y[index], y[index-1]
    zi1, zi0 = z[index], z[index-1]
    hi1 = xi1 - xi0

    # calculate cubic
    f0 = zi0/(6*hi1)*(xi1-x0)**3 + \
         zi1/(6*hi1)*(x0-xi0)**3 + \
         (yi1/hi1 - zi1*hi1/6)*(x0-xi0) + \
         (yi0/hi1 - zi0*hi1/6)*(xi1-x0)
    return f0


# plotting the original line
x1 = t_seq
y1 = zr_seq
plt.plot(x1, y1, 'o',label = "before")

# plotting the line after NS line
x2=np.arange(1/12, 30.01, 1/12)
y2 = NS_pred(zr_seq)
plt.plot(x2,y2, label = "afer NS")

# plotting the line after cubic spline line
x3=np.arange(1/12, 30.01, 1/12)
y3=cubic_interp1d(y1)
plt.plot(x3,y3, label = "afer cubic spline")

plt.title('CS vs NS')
# show a legend on the plot
plt.legend()
# Display a figure.
plt.show()
```



In [11]:

```
# NS interpolated zr dataframe in months
NS_zr_df = np.array([])
for i in range(len(b_df)):
    y = b_df[i]
    b0 = NS_pred(y)
    NS_zr_df=np.append(NS_zr_df, b0, axis=0)

NS_zr_df.shape=(528,360)
df_NS_zr=pd.DataFrame(NS_zr_df)
df_NS_zr.columns = np.arange(1, 361,1).tolist()

df_NS_zr
```

Out[11]:

	1	2	3	4	5	6	7	8	9
0	-0.009803	-0.008499	-0.007208	-0.005928	-0.004661	-0.003406	-0.002163	-0.000931	0.
1	-0.011248	-0.009926	-0.008616	-0.007318	-0.006033	-0.004759	-0.003498	-0.002248	-0.
2	-0.011367	-0.010091	-0.008826	-0.007573	-0.006331	-0.005100	-0.003880	-0.002672	-0.
3	-0.011290	-0.010047	-0.008816	-0.007596	-0.006387	-0.005190	-0.004003	-0.002828	-0.
4	-0.011836	-0.010582	-0.009340	-0.008110	-0.006891	-0.005683	-0.004487	-0.003302	-0.
5	-0.012267	-0.011021	-0.009786	-0.008562	-0.007350	-0.006148	-0.004957	-0.003777	-0.
6	-0.001950	-0.001655	-0.001312	-0.000925	-0.000496	-0.000028	0.000478	0.001018	0.
7	-0.011190	-0.009958	-0.008737	-0.007526	-0.006325	-0.005135	-0.003956	-0.002787	-0.
8	-0.011510	-0.010279	-0.009060	-0.007851	-0.006653	-0.005465	-0.004288	-0.003122	-0.
9	-0.004089	-0.003560	-0.003002	-0.002416	-0.001805	-0.001169	-0.000512	0.000167	0.
10	-0.006023	-0.005222	-0.004408	-0.003581	-0.002742	-0.001893	-0.001034	-0.000166	0.

11	-0.006911	-0.005990	-0.005061	-0.004126	-0.003185	-0.002239	-0.001289	-0.000334	0.
12	-0.007759	-0.006723	-0.005683	-0.004639	-0.003592	-0.002543	-0.001492	-0.000440	0.
13	-0.007972	-0.006932	-0.005884	-0.004831	-0.003773	-0.002711	-0.001645	-0.000576	0.
14	-0.006208	-0.005315	-0.004402	-0.003471	-0.002525	-0.001563	-0.000588	0.000398	0.
15	-0.004554	-0.003869	-0.003154	-0.002411	-0.001641	-0.000848	-0.000032	0.000805	0.
16	-0.006160	-0.005292	-0.004408	-0.003508	-0.002593	-0.001665	-0.000726	0.000224	0.
17	-0.004457	-0.003786	-0.003086	-0.002359	-0.001607	-0.000831	-0.000034	0.000782	0.
18	0.001504	0.001405	0.001388	0.001450	0.001584	0.001786	0.002052	0.002376	0.
19	-0.003002	-0.002557	-0.002077	-0.001564	-0.001020	-0.000446	0.000154	0.000780	0.
20	-0.001399	-0.001146	-0.000849	-0.000512	-0.000136	0.000275	0.000720	0.001197	0.
21	-0.002646	-0.002199	-0.001724	-0.001223	-0.000699	-0.000152	0.000417	0.001005	0.
22	-0.002034	-0.001677	-0.001286	-0.000861	-0.000406	0.000077	0.000588	0.001124	0.
23	-0.004430	-0.003838	-0.003230	-0.002607	-0.001970	-0.001320	-0.000658	0.000016	0.
24	-0.005131	-0.004507	-0.003868	-0.003212	-0.002543	-0.001861	-0.001166	-0.000460	0.
25	-0.002934	-0.002557	-0.002152	-0.001719	-0.001260	-0.000777	-0.000272	0.000255	0.
26	-0.001014	-0.000812	-0.000577	-0.000309	-0.000011	0.000316	0.000670	0.001050	0.
27	0.000338	0.000399	0.000504	0.000650	0.000834	0.001056	0.001311	0.001600	0.
28	-0.000516	-0.000389	-0.000222	-0.000019	0.000220	0.000492	0.000795	0.001128	0.
29	0.000239	0.000288	0.000378	0.000505	0.000669	0.000866	0.001097	0.001358	0.
...	...	...	...	...	...	...	...	...	
498	0.021775	0.021981	0.022186	0.022390	0.022593	0.022794	0.022995	0.023194	0.
499	0.022052	0.022254	0.022454	0.022653	0.022851	0.023048	0.023244	0.023439	0.
500	0.022103	0.022274	0.022446	0.022616	0.022785	0.022954	0.023122	0.023290	0.
501	0.019987	0.020119	0.020250	0.020381	0.020511	0.020642	0.020771	0.020901	0.
502	0.020349	0.020447	0.020544	0.020642	0.020740	0.020837	0.020934	0.021031	0.
503	0.020085	0.020179	0.020272	0.020366	0.020459	0.020553	0.020646	0.020739	0.
504	0.020104	0.020176	0.020248	0.020320	0.020393	0.020465	0.020538	0.020611	0.
505	0.019434	0.019520	0.019607	0.019694	0.019780	0.019867	0.019953	0.020040	0.
506	0.021182	0.021315	0.021448	0.021580	0.021712	0.021843	0.021974	0.022104	0.
507	0.021744	0.021883	0.022022	0.022160	0.022298	0.022435	0.022571	0.022707	0.
508	0.020690	0.020818	0.020946	0.021073	0.021199	0.021325	0.021450	0.021575	0.
509	0.018518	0.018646	0.018774	0.018901	0.019028	0.019154	0.019280	0.019405	0.

510	0.018559	0.018688	0.018817	0.018945	0.019073	0.019201	0.019328	0.019454	0.
511	0.018638	0.018814	0.018989	0.019163	0.019336	0.019508	0.019679	0.019849	0.
512	0.018833	0.019010	0.019185	0.019360	0.019533	0.019706	0.019878	0.020048	0.
513	0.018460	0.018646	0.018831	0.019015	0.019198	0.019380	0.019561	0.019740	0.
514	0.018812	0.019015	0.019217	0.019418	0.019618	0.019816	0.020013	0.020209	0.
515	0.018031	0.018259	0.018486	0.018711	0.018935	0.019157	0.019377	0.019596	0.
516	0.018331	0.018522	0.018712	0.018900	0.019087	0.019274	0.019459	0.019643	0.
517	0.018938	0.019117	0.019296	0.019473	0.019649	0.019825	0.019999	0.020172	0.
518	0.018438	0.018634	0.018829	0.019022	0.019214	0.019405	0.019595	0.019783	0.
519	0.018598	0.018811	0.019023	0.019234	0.019443	0.019650	0.019856	0.020060	0.
520	0.017211	0.017487	0.017759	0.018030	0.018298	0.018563	0.018826	0.019086	0.
521	0.017303	0.017568	0.017831	0.018092	0.018350	0.018606	0.018860	0.019112	0.
522	0.017069	0.017322	0.017573	0.017822	0.018070	0.018315	0.018558	0.018800	0.
523	0.017542	0.017764	0.017986	0.018205	0.018424	0.018641	0.018856	0.019070	0.
524	0.017492	0.017717	0.017940	0.018162	0.018382	0.018601	0.018818	0.019033	0.
525	0.017095	0.017304	0.017511	0.017717	0.017921	0.018124	0.018326	0.018526	0.
526	0.016531	0.016694	0.016856	0.017017	0.017177	0.017336	0.017494	0.017652	0.
527	0.016631	0.016795	0.016958	0.017121	0.017282	0.017443	0.017603	0.017762	0.

528 rows × 360 columns

In [66]:
```
df_NS_zr.to_csv('NS interpolated spot rate.csv', index=False)
```

2. Compute Annual Volatility of historical spot rate for each maturity

In [68]:
```
# Annual Volatility
timewindow=260
def estVol(timewindow):
    df=df_NS_zr[527-timewindow:]
    Vols=[]
    # Iterate over the index range from o to max number of columns in
dataframe
    for i in df.columns:
        zr= df[i][:-1].values
        zr_lag=df[i][1:].values
        change=[zl/z for zl,z in zip(zr_lag,zr)]
        log_change=np.log(change)
        Vol=np.nanstd(log_change)
        #calculate a list of vols for each maturiy
        Vols= Vols+[Vol]
    return (Vols)
annual_vol=estVol(timewindow)
```

```
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:11: Ru
ntimeWarning: invalid value encountered in log
  # This is added back by InteractiveShellApp.init_path()
```

In [69]:
```
#put Vols into a dataframe
annual_vol= np.array(annual_vol)*math.sqrt(52)
annual_vol.shape=(1,360)
df_Vols=pd.DataFrame(annual_vol)
df_Vols.columns = np.arange(1, 361,1).tolist()
df_Vols
```

Out[69]:

	1	2	3	4	5	6	7	8	9	10
0	2.006733	2.065997	1.481464	1.49103	1.543008	1.633581	2.776764	1.01976	0.65664	0.53

1 rows × 360 columns

In [70]:
```
df_Vols.to_csv('historical vols for each month based on 260 timewindow
.csv', index=False)
```

Installment 3

1. interest rate option Vasicek pricing model

1.1 analytic method

In [14]:
```
real_r=df_bst.iloc[-1].values   # marketable maturity spot rate from m
ost recent week

T_univ=[1/12,0.25,0.5,1,2,3,5,7,10,20,30]
real_p= [math.exp(-r*T) for r,T in zip(real_r,T_univ)] # caluate price
of 1 month using 1month rate and T=1


x0=[0.001,0.001,0.017,0.0156]          # a,b,sigma,r0

# given a,b,sigma,r0, compute SSE
def objective_f(x):

    B0= [(1- math.exp(-x[0]*T)) /x[0] for T in T_univ]
    A0= [math.exp((b0-T)*(x[0]**2*x[1]-x[2]**2/2)/(x[0]**2) -  x[2]**2
* b0**2 /4/x[0]) for b0,T in zip(B0,T_univ)]
    model_p= [a0*math.exp(-b0 * x0[3]) for a0,b0 in zip(A0,B0)]
    model_r=[-math.log(m_p)/T for m_p,T in zip(model_p,T_univ)]

    SSE= sum([(rp-mp)**2 for rp,mp in zip(real_p,model_p)] )*1000000
    return(SSE)

#objective_f(x0)

def model(x):

    B0= [(1- math.exp(-x[0]*T)) /x[0] for T in T_univ]
    A0= [math.exp((b0-T)*(x[0]**2*x[1]-x[2]**2/2)/(x[0]**2) -  x[2]**2
* b0**2 /4/x[0]) for b0,T in zip(B0,T_univ)]
    model_p= [a0*math.exp(-b0 * x0[3]) for a0,b0 in zip(A0,B0)]
    model_r=[-math.log(m_p)/T for m_p,T in zip(model_p,T_univ)]
    return(model_p,model_r)
```

In [15]:
```
# minimizing SSE using optimization to get opt params.

bnds = ((0.001, None), (0.001, None), (0.001, None),(None,None))
sol = minimize(objective_f, x0,bounds=bnds)
best_param=sol.x

print(best_param,objective_f(best_param))
```

```
[0.0783922  0.0732067  0.01336035 0.0156    ] 101.77480610642078
```

In [16]:
```
# V analytic !!!SLIDES!!! method
x0=[0.001,0.001,0.017,0.0156]

S=3             # option maturity
T=5             # Bond maturity
X=0.57           # strike price

def E_call_Vas_ana(x0,S,T,X):

    B0= [(1- math.exp(-best_param[0]*T)) /best_param[0] for T in T_uni
v]
    # SSE using best parameters
    best_SSE=objective_f(best_param)

    #model price and model rate with best_param
    model_p=model(best_param)[0]
    model_r=model(best_param)[1]
    model_p[1]
    # make T_univ and model_p as dictionary
    dict_p = dict(zip(T_univ, model_p))
    # make T_univ and model_r as dictionary
    dict_r = dict(zip(T_univ, model_r))
    # make T_univ and model_B as dictionary
    dict_b = dict(zip(T_univ, B0))

    # caculate sig_hat, d using formulas
    sig_hat=best_param[2]/best_param[0]*(1-math.exp(-best_param[0]*(T-
S))) * math.sqrt((1-math.exp(-2*best_param[0]*S))/2/best_param[0])

    d=1/sig_hat* math.log(dict_b[T]/X * dict_b[S]) +0.5*sig_hat

    ##############################
    # price of call
    call= dict_b[T]*norm.cdf(d)-X*dict_b[S]*norm.cdf(d-sig_hat)

    print('best parameters: ',best_param)
    print('best_SSE: ',best_SSE)
    print ('call price: ',call)


#E_call_Vas_ana(x0,S,T,X)
```

In [17]:

```
# V analytic EXCEL method
x0=[0.001,0.001,0.017,0.0156]

S=3             # option maturity
T=5             # Bond maturity
X=0.57           # strike price

def E_call_Vas_ana(x0,S,T,X):

    B0= [(1- math.exp(-best_param[0]*T)) /best_param[0] for T in T_uni
v]
    # SSE using best parameters
    best_SSE=objective_f(best_param)

    #model price and model rate with best_param
    model_p=model(best_param)[0]
    model_r=model(best_param)[1]
    model_p[1]
    # make T_univ and model_p as dictionary
    dict_p = dict(zip(T_univ, model_p))
    # make T_univ and model_r as dictionary
    dict_r = dict(zip(T_univ, model_r))
    # make T_univ and model_B as dictionary
    dict_b = dict(zip(T_univ, B0))

    # caculate sig_hat, d using formulas
    sig_hat=best_param[2]/best_param[0]/math.sqrt(S)*(1-math.exp(-best
_param[0]*(T-S)))*math.sqrt((1-math.exp(-2*best_param[0]*S))/2/best_pa
ram[0])

    d1=(math.log(dict_p[T]/X)+ (dict_r[S]+0.5*sig_hat**2)*S)/sig_hat/m
ath.sqrt(S)
    d2=d1-sig_hat*math.sqrt(S)

    ##############################
    # price of call
    call= dict_p[T]*norm.cdf(d1)-X*dict_p[S]*norm.cdf(d2)

    print('best parameters: ',best_param)
    print('best_SSE: ',best_SSE)
    print ('call price: ',call)


E_call_Vas_ana(x0,S,T,X)
```

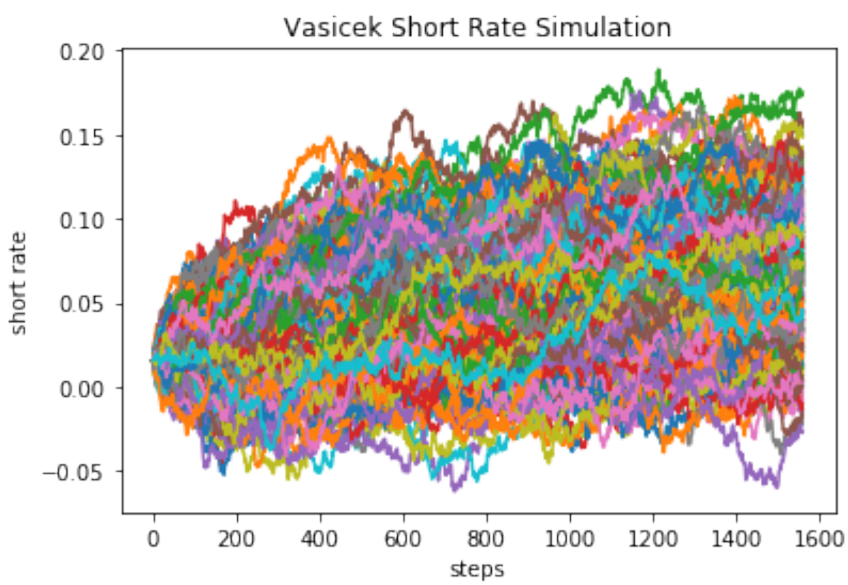
```
best parameters:  [0.0783922  0.0732067  0.01336035 0.0156    ]
best_SSE:  101.77480610642078
call price:  0.34838800569248773
```

1.2 simulation method

Excel Formual

In [72]:
```
a= 0.0783922          # mean reverting parameter
b= 0.0732067          # long term mean rate
sigma=0.01336035      # volatility
r0= 0.0156           #   initial spot rate

n=52*30             # number of simulation for 30 years

# generate interest list given a,b,r0 and sigma  !!!EXCEL!
def r_generator (a,b,r0,sigma,n): #Creates an array of r_t (discrete)
    i = 0
    r = [r0]
    dt= 1/52
    for i in range(n): #goes from i = 0 to (numb_its - 1)
        brownian = random.normalvariate(0, 1) #Brownian motion variabl
e
        r_new = r[i] + a*(b-r[i])*dt + sigma*sqrt(dt)*brownian #calcul
ates each r_t
        r.append(r_new)
    return r

#r_generator (a,b,r0,sigma,n)
```

In [73]:
```
m= 500                 # number of paths

# generate interest list matrix
def r_matrix_generator (a,b,r0,sigma,n,m):
    r_m= []
    for i in range (m):
        r_l=r_generator (a,b,r0,sigma,n)
        r_m= r_m + [r_l]
    return(r_m)
r_m= r_matrix_generator (a,b,r0,sigma,n,m)



# short rate simulation
x= range(52*30+1)
for i in range(len(r_m)):
    y=r_m[i]
    plt.plot(x,y)

plt.title('Vasicek Short Rate Simulation')
plt.xlabel('steps')
plt.ylabel('short rate')
plt.show()
```



In [75]:
```
tau=20          # option maturity
T=30            # bond maturity
k=0.5           # strike price

def payoff(r,T,tau,k):
    if tau<=T and tau>=0: #tau is between 0 and T
        n=tau*52
        N=T*52
        bond_price_tau= math.exp(- sum(r[n:N])/52)
        payoff_tau= max(bond_price_tau-k,0)
        payoff_0= payoff_tau * math.exp(- sum(r[:n])/52)
    else:
        payoff_0=0
    return (payoff_0)


payoff(r_m[-1],T,tau,k)
```

Out[75]: 0.07318848627877567

In [22]:
```
a= 0.0783922         # mean reverting parameter
b= 0.0732067         # long term mean rate
sigma=0.01336035     # volatility
r0= 0.0156           # initial spot rate
m= 500               # number of paths
n=52*30            # number of simulation for 30 years

r_m= r_matrix_generator (a,b,r0,sigma,n,m) # m*n simulated spot rate m
atrix

tau=3          # option maturity
T=5            # bond maturity
k=0.57           # strike price


def E_call_Vas_sim(T,tau,k,r_m):
    payoffs_0=[payoff(r,T,tau,k) for r in r_m]
    call = sum(payoffs_0)/m
    return (call)

E_call_Vas_sim(T,tau,k,r_m)
```

Out[22]: 0.350099532167443

2. BDT pricing model

In [23]:
```
#given implied vol, and min rate
def get_r_leafs(X):
    r0 = X[1]
    r_leafs = [r0]
    for i in range(T):
        r_up = r0*(np.exp(2*X[0]*(h**0.5)))
        r_leafs.append(r_up)
        r0 = r_up

    return r_leafs

#r_leafs=get_r_leafs(X)
#print(r_leafs,len(r_leafs))
```

In [24]:
```
#given r_tree, get bond price tree
def get_b_tree(r_leafs):
    b_tree = [[1]*(len(r_leafs)+1)]
    b_leafs = []
    for j in range(len(r_leafs)):
        b_price = 0.5*np.exp(-r_leafs[j]*h)*(b_tree[-1][j]+b_tree[-1][
j+1])
        b_leafs.append(b_price)
    b_tree.append(b_leafs)

    for i in range(1,len(r_leafs)):
        b_leafs = []
        for j in range(len(r_leafs)-i):
            b_price = 0.5*np.exp(-r_tree[-i][j]*h)*(b_tree[-1][j]+b_tr
ee[-1][j+1])
            b_leafs.append(b_price)
        b_tree.append(b_leafs)


    return b_tree
#b_tree=get_b_tree(r_leafs)
#print(len(b_tree),len(b_tree[0]),len(b_tree[1]),b_tree)
```

In [25]:
```
# minimize SSE of price and vol
def objective_BDT(x):


    r_leafs = get_r_leafs(x)
    b_tree = get_b_tree(r_leafs)

    model_price = b_tree[-1][0]
    se_p = (model_price - market_price[T])**2

    R_up = -np.log(b_tree[-2][1])/(h*(T))
    R_down = -np.log(b_tree[-2][0])/(h*(T))
    model_vol = np.log(R_up/R_down)/2*(1/h)**0.5
    se_vol = (model_vol - annual_vol[T])**2

    SSE = (se_p + se_vol)*10000

    return SSE
#objective_BDT(x)
```

In [26]:
```
## Call Option
S = 3 #Option Maturity
T = 5 #Bond Maturity
K = 0.57#Strike Price

spot_rate=NS_zr_df[-1]
monthly_ttm=df_NS_zr.columns
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
annual_vol=df_Vols.values[0].tolist()
```

In [27]:
```
## Solve interest rate tree and bond price tree

from scipy.optimize import minimize
r_tree = [[spot_rate[0]]]
TP = T*12
h = 1/12
X = [0.33,0.01]
BDT_set = []
SSE_set = []

for T in range(1,int(TP)):
    sol = minimize(objective_BDT, X, bounds=((0.00001, None), (0.00001
, None)))
    param = list(sol.x)
    SSE_set.append(objective_BDT(param))
    BDT_set.append(param)
    X[0] = annual_vol[T+2]
    X[1] = param[1]
    l = get_r_leafs(param)
    r_tree.append(l)
```

In [28]:
```
## Calculate option price

r_tree = r_tree[:-1]
b_tree = get_b_tree(l)

option_price = [[]]
OP = S/h

for j in range(len(b_tree[-int(OP)-1])):
    op = max(b_tree[-int(OP)-1][j] - K,0)
    option_price[0].append(op)

for k in range(1,int(OP)+1):
    op_leafs = []
    for i in range(len(option_price[0])-k):
        op = (option_price[0][i]+option_price[0][i+1])/2*np.exp(-r_tre
e[int(OP)-k][i]*h)
        op_leafs.append(op)
    option_price.append(op_leafs)

print('BDT',option_price[-1][0])
```

```
BDT 0.4134520914323181
```

In [29]:
```
real_r=df_bst.iloc[-1].values
x0=[0.001,0.001,0.017,0.0156]

S=3             # option maturity
T=5             # Bond maturity
X=0.57           # strike price

E_call_Vas_ana(x0,S,T,X)
```

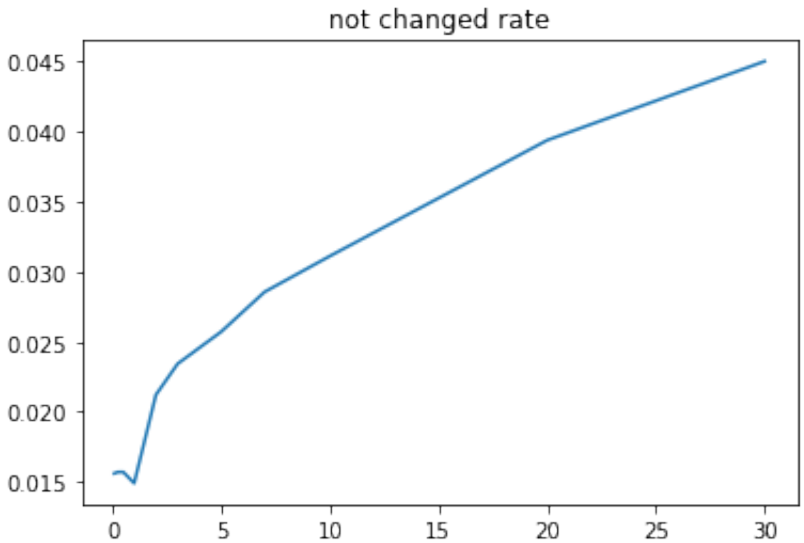
```
best parameters:  [0.0783922  0.0732067  0.01336035 0.0156    ]
best_SSE:  101.77480610642078
call price:  0.34838800569248773
```

In [30]:
```
a= 0.0783922          # mean reverting parameter
b= 0.0732067          # long term mean rate
sigma=0.01336035      # volatility
r0= 0.0156            # initial spot rate
m= 500                # number of paths
n=52*30             # number of simulation for 30 years

r_m= r_matrix_generator (a,b,r0,sigma,n,m) # m*n simulated spot rate m
atrix


E_call_Vas_sim(T,S,X,r_m)
```

Out[30]:
```
0.346019160303399
```

Vasieck Sensitivity analysis

In [31]:
```
# V marketable rates of last week
real_r
x= [1/12,0.25,0.5,1,2,3,5,7,10,20,30]
plt.plot(x,real_r)
plt.title('not changed rate')
plt.show()
```



In [32]:
```
real_p= [math.exp(-r*T) for r,T in zip(real_r,T_univ)] # caluate price of 1 month using 1month rate and T=1
x0=[0.001,0.001,0.01,0.0156]

# minimizing SSE using optimization to get opt params.

bnds = ((0.001, None), (0.001, None), (0.001, None),(None,None))
sol = minimize(objective_f, x0,bounds=bnds)
best_param=sol.x

print(best_param,objective_f(best_param))
E_call_Vas_ana(x0,S,T,X)
```

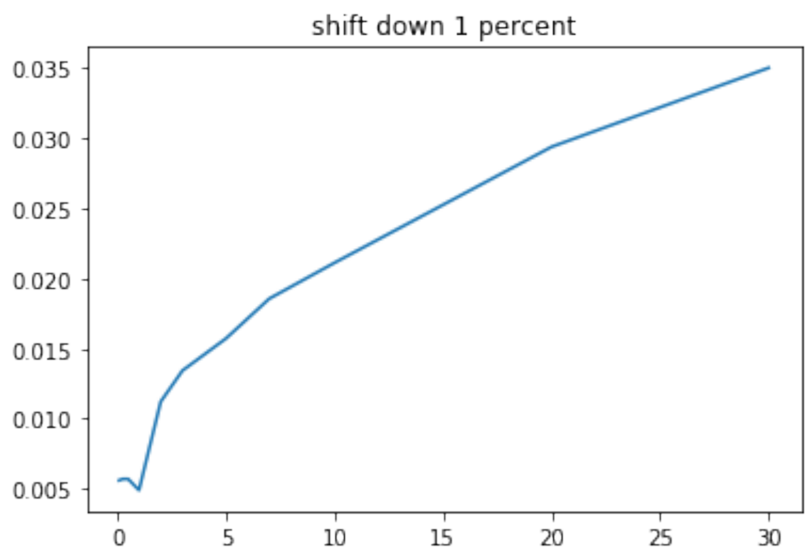
```
[0.07839222 0.07320668 0.01336035 0.0156    ] 101.7748061095605
best parameters:  [0.07839222 0.07320668 0.01336035 0.0156    ]
best_SSE:  101.7748061095605
call price:  0.3483880042743772
```

In [33]:
```
a= best_param[0]         # mean reverting parameter
b= best_param[1]         # long term mean rate
sigma=best_param[2]     # volatility
r0=best_param[3]           # initial spot rate
m= 500                 # number of paths
n=52*30              # number of simulation for 30 years

r_m= r_matrix_generator (a,b,r0,sigma,n,m) # m*n simulated spot rate m
atrix


E_call_Vas_sim(T,S,X,r_m)
```

Out[33]: 0.3464519958214482

marketable rates of last week shift down by 100 basis points

In [34]:
```
real_r_d= [r -0.01 for r in real_r]
x= [1/12,0.25,0.5,1,2,3,5,7,10,20,30]
plt.plot(x,real_r_d)
plt.title('shift down 1 percent')
plt.show()
```



In [35]:
```
real_p= [math.exp(-r*T) for r,T in zip(real_r_d,T_univ)] # caluate pri
ce of 1 month using 1month rate and T=1
x0=[0.001,0.001,0.01,0.0056]

# minimizing SSE using optimization to get opt params.

bnds = ((0.001, None), (0.001, None), (0.001, None),(None,None))
sol = minimize(objective_f, x0,bounds=bnds)
best_param=sol.x

print(best_param,objective_f(best_param))
E_call_Vas_ana(x0,S,T,X)
```

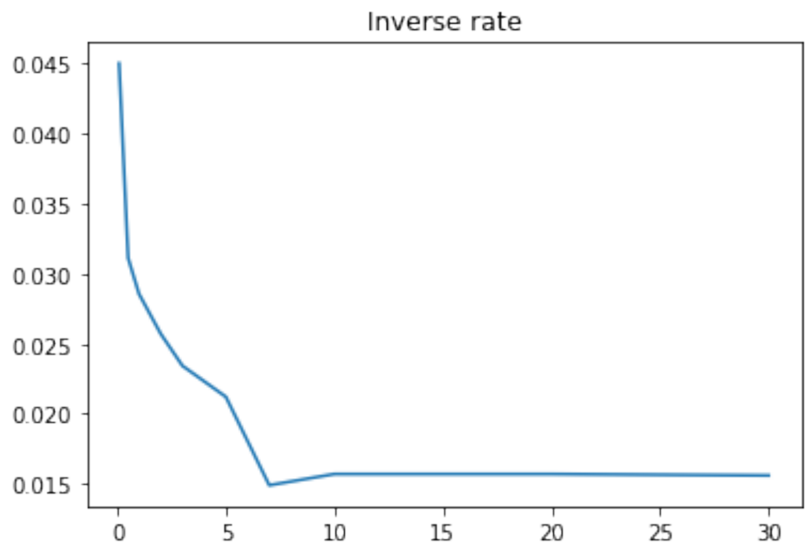
```
[0.07628458 0.06399164 0.01308083 0.0056    ] 123.21321609235048
best parameters:  [0.07628458 0.06399164 0.01308083 0.0056    ]
best_SSE:  123.21321609235048
call price:  0.3776526053232837
```

In [36]:
```
a= best_param[0]        # mean reverting parameter
b= best_param[1]        # long term mean rate
sigma=best_param[2]    # volatility
r0=best_param[3]          # initial spot rate
m= 500              # number of paths
n=52*30           # number of simulation for 30 years

r_m= r_matrix_generator (a,b,r0,sigma,n,m) # m*n simulated spot rate m
atrix


E_call_Vas_sim(T,S,X,r_m)
```

Out[36]:
```
0.37303182587651457
```

V inverted marketable rates for last week

In [37]:
```
real_r_inv= np.flipud(real_r)
x= [1/12,0.25,0.5,1,2,3,5,7,10,20,30]
plt.plot(x,real_r_inv)
plt.title('Inverse rate')
plt.show()
```



In [38]:
```
real_p= [math.exp(-r*T) for r,T in zip(real_r_inv,T_univ)] # caluate price of 1 month using 1month rate and T=1
x0=[0.001,0.001,0.01,0.045]

# minimizing SSE using optimization to get opt params.

bnds = ((0.001, None), (0.001, None), (0.001, None),(None,None))
sol = minimize(objective_f, x0,bounds=bnds)
best_param=sol.x

print(best_param,objective_f(best_param))
E_call_Vas_ana(x0,S,T,X)
```

```
[1.55701455 0.01470654 0.00209884 0.045      ] 548.5251057909801
best parameters:  [1.55701455 0.01470654 0.00209884 0.045      ]
best_SSE:  548.5251057909801
call price:  0.37622622554558083
```

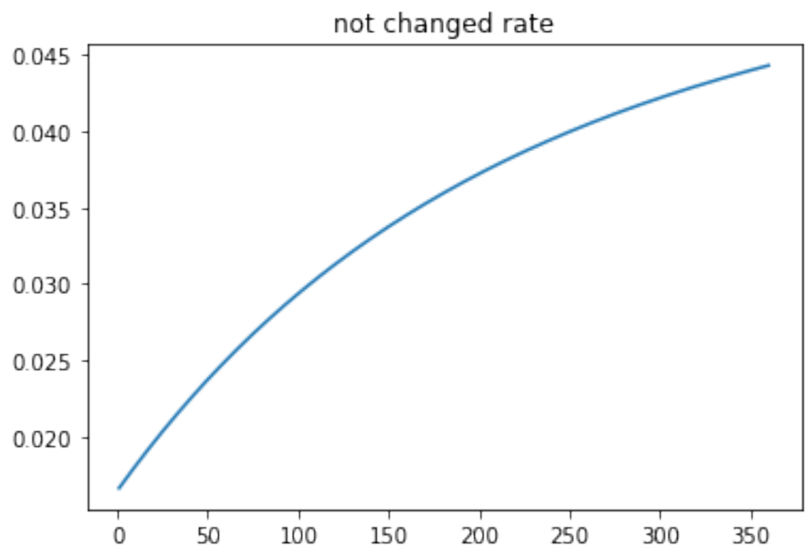
In [39]:
```
a= best_param[0]          # mean reverting parameter
b= best_param[1]          # long term mean rate
sigma=best_param[2]     # volatility
r0=best_param[3]           # initial spot rate
m= 500                 # number of paths
n=52*30              # number of simulation for 30 years

r_m= r_matrix_generator (a,b,r0,sigma,n,m) # m*n simulated spot rate m
atrix


E_call_Vas_sim(T,S,X,r_m)
```

Out[39]: 0.3762032578253521

BDT sensitivity analysis

BDT change spot rate curve

In [40]:
```
# V marketable rates of last week
spot_rate
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
plt.plot(monthly_ttm,spot_rate)
plt.title('not changed rate')
plt.show()
```



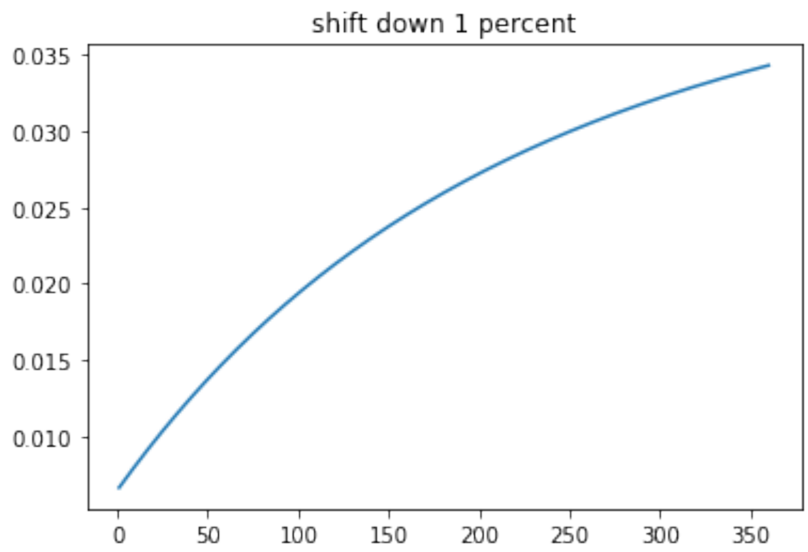
In [41]:
```
## Call Option
S = 3 #Option Maturity
T = 5 #Bond Maturity
K = 0.57#Strike Price

spot_rate=spot_rate
monthly_ttm=df_NS_zr.columns
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
annual_vol=df_Vols.values[0].tolist()
```

In [42]:
```
## Solve interest rate tree and bond price tree

from scipy.optimize import minimize
r_tree = [[spot_rate[0]]]
TP = T*12
h = 1/12
X = [0.33,0.01]
BDT_set = []
SSE_set = []

for T in range(1,int(TP)):
    sol = minimize(objective_BDT, X, bounds=((0.00001, None), (0.00001
, None)))
    param = list(sol.x)
    SSE_set.append(objective_BDT(param))
    BDT_set.append(param)
    X[0] = annual_vol[T+2]
    X[1] = param[1]
    l = get_r_leafs(param)
    r_tree.append(l)
```

In [43]:
```
## Calculate option price

r_tree = r_tree[:-1]
b_tree = get_b_tree(l)

option_price = [[]]
OP = S/h

for j in range(len(b_tree[-int(OP)-1])):
    op = max(b_tree[-int(OP)-1][j] - K,0)
    option_price[0].append(op)

for k in range(1,int(OP)+1):
    op_leafs = []
    for i in range(len(option_price[0])-k):
        op = (option_price[0][i]+option_price[0][i+1])/2*np.exp(-r_tre
e[int(OP)-k][i]*h)
        op_leafs.append(op)
    option_price.append(op_leafs)

print('BDT',option_price[-1][0])
```

```
BDT 0.4134520914323181
```

In [44]:
```
spot_rate_d= [r -0.01 for r in spot_rate]
plt.plot(monthly_ttm,spot_rate_d)
plt.title('shift down 1 percent')
plt.show()
```



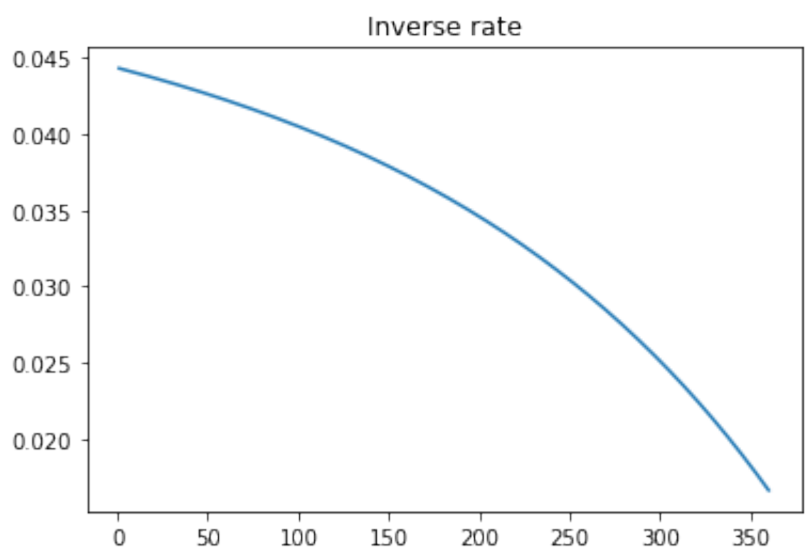
In [45]:
```
## Call Option
S = 3 #Option Maturity
T = 5 #Bond Maturity
K = 0.57#Strike Price

spot_rate=spot_rate_d
monthly_ttm=df_NS_zr.columns
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
annual_vol=df_Vols.values[0].tolist()
```

In [46]:
```
## Solve interest rate tree and bond price tree

from scipy.optimize import minimize
r_tree = [[spot_rate[0]]]
TP = T*12
h = 1/12
X = [0.33,0.01]
BDT_set = []
SSE_set = []

for T in range(1,int(TP)):
    sol = minimize(objective_BDT, X, bounds=((0.00001, None), (0.00001
, None)))
    param = list(sol.x)
    SSE_set.append(objective_BDT(param))
    BDT_set.append(param)
    X[0] = annual_vol[T+2]
    X[1] = param[1]
    l = get_r_leafs(param)
    r_tree.append(l)
```

In [47]:
```
## Calculate option price

r_tree = r_tree[:-1]
b_tree = get_b_tree(l)

option_price = [[]]
OP = S/h

for j in range(len(b_tree[-int(OP)-1])):
    op = max(b_tree[-int(OP)-1][j] - K,0)
    option_price[0].append(op)

for k in range(1,int(OP)+1):
    op_leafs = []
    for i in range(len(option_price[0])-k):
        op = (option_price[0][i]+option_price[0][i+1])/2*np.exp(-r_tre
e[int(OP)-k][i]*h)
        op_leafs.append(op)
    option_price.append(op_leafs)

print('BDT',option_price[-1][0])
```

```
BDT 0.4171799910327759
```

In [48]:
```
spot_rate=[r+0.01 for r in spot_rate]
spot_rate_inv= np.flipud(spot_rate)
plt.plot(monthly_ttm,spot_rate_inv)
plt.title('Inverse rate')
plt.show()
```



In [49]:
```
## Call Option
S = 3 #Option Maturity
T = 5 #Bond Maturity
K = 0.57#Strike Price

spot_rate=spot_rate_inv
monthly_ttm=df_NS_zr.columns
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
annual_vol=df_Vols.values[0].tolist()
```

In [50]:
```
## Solve interest rate tree and bond price tree

from scipy.optimize import minimize
r_tree = [[spot_rate[0]]]
TP = T*12
h = 1/12
X = [0.33,0.01]
BDT_set = []
SSE_set = []

for T in range(1,int(TP)):
    sol = minimize(objective_BDT, X, bounds=((0.00001, None), (0.00001
, None)))
    param = list(sol.x)
    SSE_set.append(objective_BDT(param))
    BDT_set.append(param)
    X[0] = annual_vol[T+2]
    X[1] = param[1]
    l = get_r_leafs(param)
    r_tree.append(l)
```

In [51]:

```
## Calculate option price

r_tree = r_tree[:-1]
b_tree = get_b_tree(l)

option_price = [[]]
OP = S/h

for j in range(len(b_tree[-int(OP)-1])):
    op = max(b_tree[-int(OP)-1][j] - K,0)
    option_price[0].append(op)

for k in range(1,int(OP)+1):
    op_leafs = []
    for i in range(len(option_price[0])-k):
        op = (option_price[0][i]+option_price[0][i+1])/2*np.exp(-r_tre
e[int(OP)-k][i]*h)
        op_leafs.append(op)
    option_price.append(op_leafs)

print('BDT',option_price[-1][0])
```

```
BDT 0.416162537693934
```

BDT change time window

In [52]:
```
# Annual Volatility
timewindow=130
def estVol(timewindow):
    df=df_NS_zr[527-timewindow:]
    Vols=[]
    # Iterate over the index range from o to max number of columns in dataframe
    for i in df.columns:
        zr= df[i][:-1].values
        zr_lag=df[i][1:].values
        change=[zl/z for zl,z in zip(zr_lag,zr)]
        log_change=np.log(change)
        Vol=np.nanstd(log_change)
        #calculate a list of vols for each maturiy
        Vols= Vols+[Vol]
    return (Vols)
annual_vol=estVol(timewindow)
```

```
//anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:11: Ru
ntimeWarning: invalid value encountered in log
  # This is added back by InteractiveShellApp.init_path()
```

In [53]:
```
#put Vols into a dataframe
annual_vol= np.array(annual_vol)*math.sqrt(52)
annual_vol.shape=(1,360)
df_Vols=pd.DataFrame(annual_vol)
df_Vols.columns = np.arange(1, 361,1).tolist()
df_Vols
```

Out[53]:

	1	2	3	4	5	6	7	8	9	1(
0	2.286856	2.217548	0.947448	0.649992	0.492464	0.393758	0.327711	0.282465	0.25151	0.

1 rows × 360 columns

In [54]:
```
## Call Option
S = 3 #Option Maturity
T = 5 #Bond Maturity
K = 0.57#Strike Price

spot_rate=spot_rate_d
monthly_ttm=df_NS_zr.columns
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
annual_vol=df_Vols.values[0].tolist()
```

In [55]:
```
## Solve interest rate tree and bond price tree

from scipy.optimize import minimize
r_tree = [[spot_rate[0]]]
TP = T*12
h = 1/12
X = [0.33,0.01]
BDT_set = []
SSE_set = []

for T in range(1,int(TP)):
    sol = minimize(objective_BDT, X, bounds=((0.00001, None), (0.00001
, None)))
    param = list(sol.x)
    SSE_set.append(objective_BDT(param))
    BDT_set.append(param)
    X[0] = annual_vol[T+2]
    X[1] = param[1]
    l = get_r_leafs(param)
    r_tree.append(l)
```

In [56]:
```
## Calculate option price

r_tree = r_tree[:-1]
b_tree = get_b_tree(l)

option_price = [[]]
OP = S/h

for j in range(len(b_tree[-int(OP)-1])):
    op = max(b_tree[-int(OP)-1][j] - K,0)
    option_price[0].append(op)

for k in range(1,int(OP)+1):
    op_leafs = []
    for i in range(len(option_price[0])-k):
        op = (option_price[0][i]+option_price[0][i+1])/2*np.exp(-r_tre
e[int(OP)-k][i]*h)
        op_leafs.append(op)
    option_price.append(op_leafs)

print('BDT',option_price[-1][0])
```

```
BDT 0.4054213490436012
```

BDT change time steps from 1 month to 1 quater (average of 4 months)

In [57]:
```
spot_rate=NS_zr_df[-1]
```

In [58]:
```
q_spot_rate=[]
for i in range(90):
    avg=sum(spot_rate[i*4:(i+1)*4])/4
    q_spot_rate=q_spot_rate +[avg]
q_spot_rate
```

Out[58]:
```
[0.016876066629091576,
 0.01752240203980808,
 0.018155180412503157,
 0.018774721158983366,
 0.019381335675310706,
 0.019975327551041158,
 0.020556992772853437,
 0.0211266199227216 8,
 0.021684490370780848,
 0.0222308784630299 6,
 0.022766051704013815,
 0.02329027093462034,
 0.02380379050512671,
 0.024306858443623795,
 0.024799716619944748,
 0.02528260090522031,
 0.025755741327179806,
 0.026219362221313682,
 0.026673682378010006,
 0.027118915185774604,
 0.02755526877064112,
 0.027982946131874513,
 0.028402145274068733,
 0.028813059335736375,
 0.029215876714485476,
 0.02961078118887607,
 0.029997952037046447,
 0.030377564152196647,
 0.03074978815501431,
 0.031114790503125718,
 0.03147273359765235,
 0.0318237758869514,
 0.03216807196761636,
 0.03250577268281156,
 0.032837025218012884,
```

```
0.03316197319422447,
0.03348075675873965,
0.03379351267351231,
0.0341003744012031,
0.03440147218896299,
0.03469693315001543,
0.03498688134309608,
0.035271437849807734,
0.035550720849946885,
0.03582484569485588,
0.03609392497885421,
0.0363580686088002,
0.036617383871833345,
0.03687197550134625,
0.03712194574123335,
0.037367394408462845,
0.037608418954016606,
0.03784511452224178,
0.038077574008656634,
0.03830588811625189,
0.03853014541032775,
0.03875043237190587,
0.03896683344975413,
0.03917943111106138,
0.03938830589079804,
0.03959353643979763,
0.03979519957159336,
0.03999337030804279,
0.04018812192377292,
0.04037952598947707,
0.040567652414094005,
0.040752569485898976,
0.04093434391253575,
0.04111304086001752,
0.041288723990724065,
0.041461455500421886,
0.041631296154333114,
0.041798305322278316,
0.04196254101291784,
0.04212405990711536,
0.04228291739044698,
0.04243916758487834,
0.04259286337963179,
0.04274405646126482,
0.042892797342980765,
0.04303913539319184,
0.043183118863354235,
0.04332479491509443,
0.04346420964664541,
0.04360140811861078,
```

```
 0.04373643437907459 6,
 0.04386933148807399,
 0.04400014154145124 5,
 0.04412890569410176,
 0.04425566418263352]
```

In [59]:

```
## Call Option
S = 3 #Option Maturity
T = 5 #Bond Maturity
K = 0.57#Strike Price

spot_rate=q_spot_rate
monthly_ttm=list(range(1,91))
market_price=[math.exp(-r*m) for r,m in zip(spot_rate,monthly_ttm)]
annual_vol=df_Vols.values[0].tolist()
```

In [60]:

```
## Solve interest rate tree and bond price tree

from scipy.optimize import minimize
r_tree = [[spot_rate[0]]]
TP = T*4
h = 1/4
X = [0.33,0.01]
BDT_set = []
SSE_set = []

for T in range(1,int(TP)):
    sol = minimize(objective_BDT, X, bounds=((0.00001, None), (0.00001
, None)))
    param = list(sol.x)
    SSE_set.append(objective_BDT(param))
    BDT_set.append(param)
    X[0] = annual_vol[T+2]
    X[1] = param[1]
    l = get_r_leafs(param)
    r_tree.append(l)
```

In [61]:
```
## Calculate option price

r_tree = r_tree[:-1]
b_tree = get_b_tree(l)

option_price = [[]]
OP = S/h

for j in range(len(b_tree[-int(OP)-1])):
    op = max(b_tree[-int(OP)-1][j] - K,0)
    option_price[0].append(op)

for k in range(1,int(OP)+1):
    op_leafs = []
    for i in range(len(option_price[0])-k):
        op = (option_price[0][i]+option_price[0][i+1])/2*np.exp(-r_tre
e[int(OP)-k][i]*h)
        op_leafs.append(op)
    option_price.append(op_leafs)

print('BDT',option_price[-1][0])
```

```
BDT 0.3542444068533473
```

In [ ]: