PROJECT 2

Carter Rhea

In [65]:

```
from ODE_IVP_module import *
from ODE_IVP_VECTOR import *
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [66]:

```
a = 0.
b = 2.*np.pi
init = 1.
K=1.
def g(t,y):
    return np.cos(t)-K*(y-np.sin(t))
```

Exact Solution of ODE:

Here I shall solve the ODE exactly so we can compare it to the numerical scheme.

$$\frac{dx}{dt} = cos(t) - kx + ksin(t)$$

$$\frac{dx}{dt} + kx = cos(t) + ksin(t)$$

Since this is linear we can solve it using the integrating factor... which is $I = e^{kt}$

$$xe^{kt} = \int e^{kt}cos(t) + ke^{kt}sin(t)dt$$

For ease I will break the integral into two parts and proceed to do integration by parts...

$$I_1 = e^{kt}sin(t) - k\int sin(t)e^{kt}dt$$

$$I_1 = e^{kt}sin(t) + ke^{kt}cos(t) - k^2\int cos(t)e^{kt}$$

$$I_1 = \frac{e^{kt}sin(t) + ke^{kt}cos(t)}{1 + k^2}$$

$$I_2 = -ke^{kt}cos(t) - k^2 \int cos(t)e^{kt}dt$$

$$I_2 = -ke^{kt}cos(t) + k^2e^{kt}sin(t) - k^3 \int sin(t)e^{kt}$$

$$I_2 = \frac{-ke^{kt}cos(t) + k^2e^{kt}sin(t)}{1+k^2}$$

Hence our integral is:

$$xe^{kt} = \frac{e^{kt}sin(t) + ke^{kt}cos(t)}{1+k^2} + \frac{-ke^{kt}cos(t) + k^2e^{kt}sin(t)}{1+k^2} + c$$

$$xe^{kt} = \frac{e^{kt}sin(t) + k^2e^{kt}sin(t)}{1+k^2} + c$$

$$xe^{kt} = e^{kt}sin(t) + c$$

Thus with a bit of division and plugging in our initial condition we obtain,

$$x = sin(t) + init * e^{-kt}$$

The search for a good step size

Analycially we know that for stability $0 < Kh < 1$ where $h = (b-a)/n$. Therefore,

$$0 < Kh < 1$$

$$0 < K\frac{b-a}{n}$$

$$n > K(b-a)$$

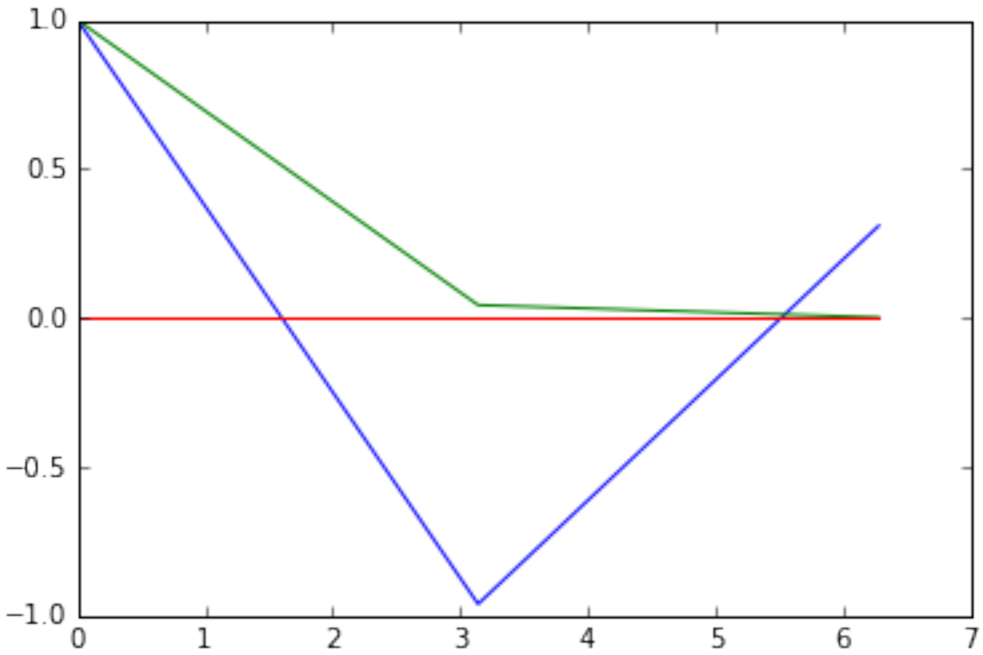
Hence for $K = 1, a = 0, and\ b = 2$ we must have $n > 2$

In [67]:

```
n = 3
steps = np.linspace(a,b,n)
def exact(t) : return np.sin(t)+ init*np.exp(-K*t)
plt.plot(steps,Runge_Kutta(g,init,a,b,n),steps,exact(steps),steps,np.sin(steps))
```

Out[67]:

```
[<matplotlib.lines.Line2D at 0x1071b9668>,
 <matplotlib.lines.Line2D at 0x1071b99b0>,
 <matplotlib.lines.Line2D at 0x106a4f780>]
```



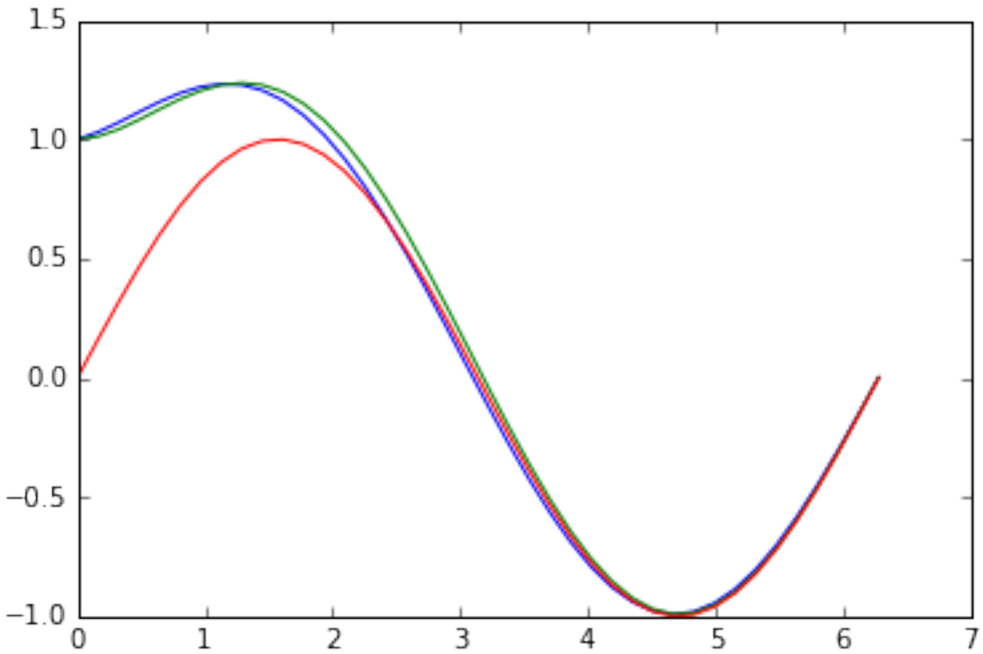
So it works, but it looks really bad!

In [73]:

```
n = 40
steps = np.linspace(a,b,n)
def exact(t) : return np.sin(t)+ init*np.exp(-K*t)
plt.plot(steps,Runge_Kutta(g,init,a,b,n),steps,exact(steps),steps,np.sin(steps))
```

Out[73]:

```
[<matplotlib.lines.Line2D at 0x1078d0320>,
 <matplotlib.lines.Line2D at 0x1078d0668>,
 <matplotlib.lines.Line2D at 0x1078d0e80>]
```

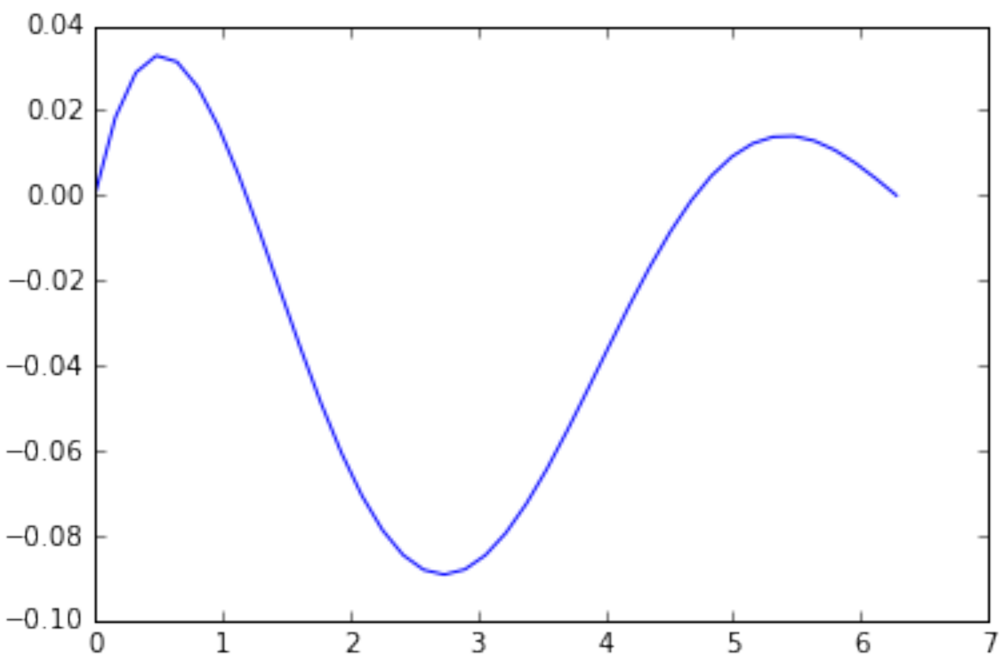


In [74]:

```
err = Runge_Kutta(g,init,a,b,n).T-exact(steps);
plt.plot(steps,err.T)
```

Out[74]:

```
[<matplotlib.lines.Line2D at 0x107ab0908>]
```



Hence we see that the error iteslef has an exponential sinusoidal form (like the exact solution)

Hence we see that the error itself has an exponential sinusoidal form (like the exact solution)

In [75]:

```
print("There is a max error of %s"%(err.max()))
```

```
There is a max error of 0.0329084269283
```

So lets go ahead and try a little bit of a better n in order to decreate the error. Here ill go ahead and try a series of n values (20,40,80,160) and plot the graphs and errors....

In [76]:

```
for n in [20,40,80,160]:
    steps = np.linspace(a,b,n)
    err = Runge_Kutta(g,init,a,b,n).T-exact(steps);
    print("There is a max error of %s with an n value of %s"%(err.max(),n))
```

```
There is a max error of 0.0493553471874 with an n value of 20
There is a max error of 0.0329084269283 with an n value of 40
There is a max error of 0.0180907668919 with an n value of 80
There is a max error of 0.0094625754892 with an n value of 160
```

The n value of 20 is reasonable. We probably wouldnt want it to be any lower since then we would not even have a decent plot

k=10

Now lets try $k = 10$ and redo all of those plots and n-value errors!

In [77]:

```
K=10.
for n in [20,40,80,160]:
    steps = np.linspace(a,b,n)
    err = Runge_Kutta(g,init,a,b,n).T-exact(steps);
    print("There is a max error of %s with an n value of %s"%(err.max(),n))
```

```
There is a max error of 13262.183118 with an n value of 20
There is a max error of 0.176995783842 with an n value of 40
There is a max error of 0.0692780684053 with an n value of 80
There is a max error of 0.0345432139179 with an n value of 160
```

In [78]:

```
## Lets check n=30 to find the stability...
steps = np.linspace(a,b,30)
err = Runge_Kutta(g,init,a,b,30).T-exact(steps);
print("There is a max error of %s with an n value of %s"%(err.max(),30))
```

```
There is a max error of 0.368648424789 with an n value of 30
```

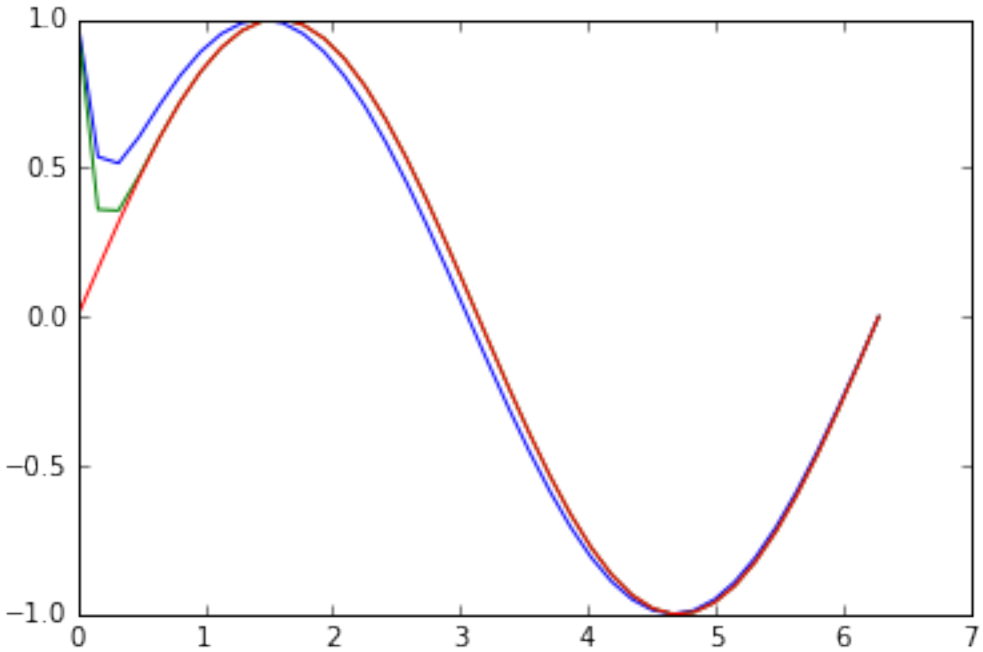
Hence it seems that an n value of about 30 is reasonable!

In [79]:

```
steps = np.linspace(a,b,40)
plt.plot(steps,Runge_Kutta(g,init,a,b,n=40),steps,exact(steps),steps,np.sin(steps))
```

Out[79]:

```
[<matplotlib.lines.Line2D at 0x107b6e668>,
 <matplotlib.lines.Line2D at 0x107b6e9b0>,
 <matplotlib.lines.Line2D at 0x107b74780>]
```



k = 100

Here we are going to use the previously used n-values and try some new ones as well!

In [51]:

```
K=100.
for n in [20,40,80,160,320,640,1280]:
    steps = np.linspace(a,b,n)
    err = Runge_Kutta(g,init,a,b,n).T-exact(steps);
    print("There is a max error of %s with an n value of %s"%(err.max(),n))
```

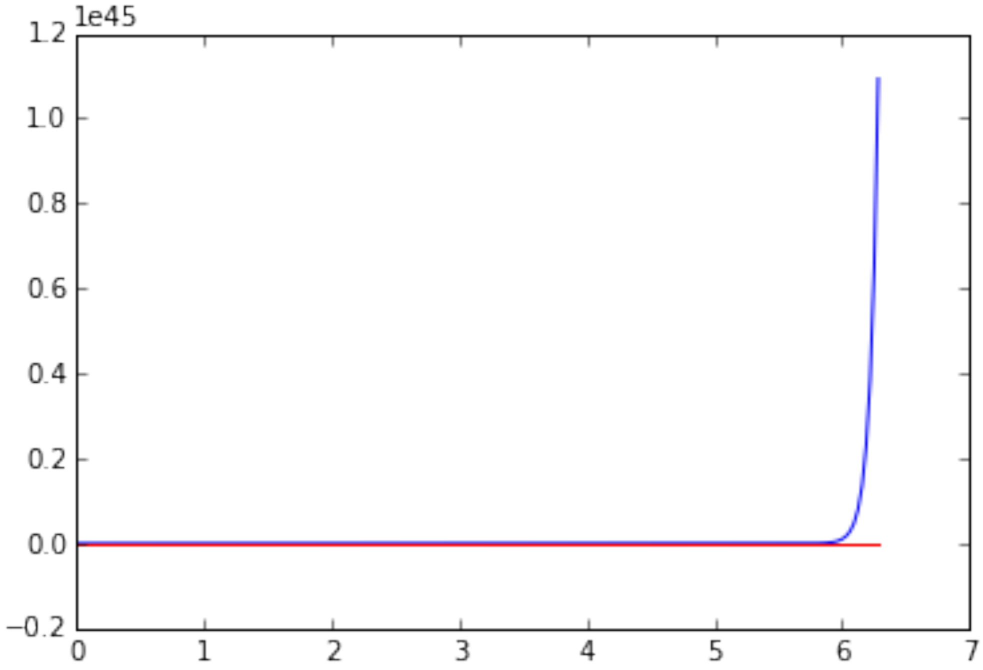
```
There is a max error of 2.41007497898e+86 with an n value of 20
There is a max error of 4.58247762164e+128 with an n value of 40
There is a max error of 3.73260751504e+158 with an n value of 80
There is a max error of 2.25057154478e+105 with an n value of 160
There is a max error of 0.195563003984 with an n value of 320
There is a max error of 0.0137096945988 with an n value of 640
There is a max error of 0.00489374563699 with an n value of 1280
```

In [52]:

```
steps = np.linspace(a,b,200)
plt.plot(steps,Runge_Kutta(g,init,a,b,n=200),steps,exact(steps),steps,np.sin(steps)
```

Out[52]:

```
[<matplotlib.lines.Line2D at 0x106dd2748>,
 <matplotlib.lines.Line2D at 0x106a62278>,
 <matplotlib.lines.Line2D at 0x106dd87f0>]
```



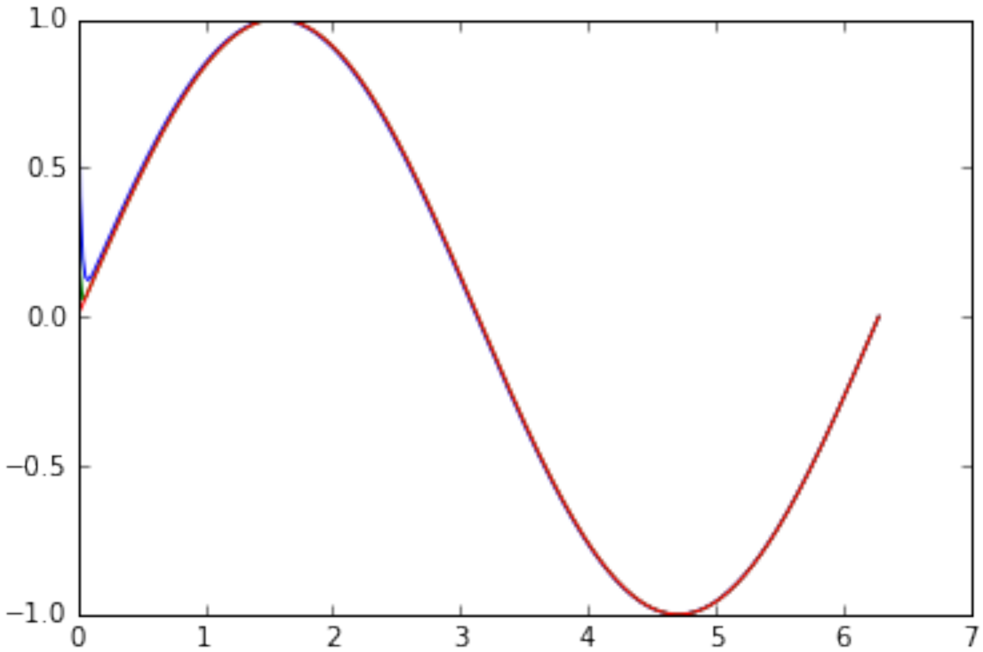
Here we can see with a K value of 100 how horrible 200 steps works! So lets move it to 300!

In [53]:

```
steps = np.linspace(a,b,300)
plt.plot(steps,Runge_Kutta(g,init,a,b,n=300),steps,exact(steps),steps,np.sin(steps)
```

Out[53]:

```
[<matplotlib.lines.Line2D at 0x106ea1470>,
 <matplotlib.lines.Line2D at 0x106de9630>,
 <matplotlib.lines.Line2D at 0x106ea1f60>]
```



Here we can see how drastically we must change n when we change the K values...

ADAMS BASHFORTH

I will be employing the Predictor-Corrector AB method using the basic algorithms defined in our book and in class.

In [54]:

```
def euler_AB(f,y_init,a,b,n):
    y = np.zeros(n)
    y[0] = y_init
    h = (b-a)/n
    t_0=a
    for i in range(1,n):
        t_i = t_0+i*h
        Pred = y[i-1]+(h/24)*(-9*f(t_i-3*h,y[i-3])+37*f(t_i-2*h,y[i-2])-59*f(t_i-h,y
        y[i] = y[i-1]+(h/24)*(f(t_i-2*h,y[i-2])-5*f(t_i-1*h,y[i-1])+19*f(t_i,y[i])+9
    return y[:]
```

In [55]:

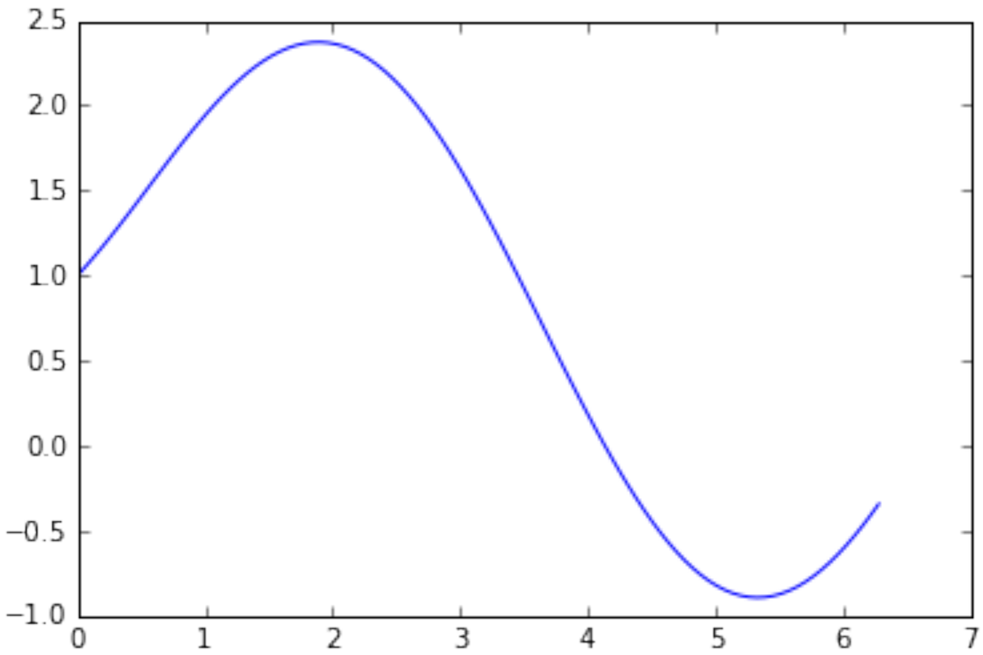
```
K = 1.
n = 100
val = euler_AB(g,init,a,b,n)
steps = np.linspace(a,b,n)
```

In [56]:

```
plt.plot(steps,val)
```

Out[56]:

```
[<matplotlib.lines.Line2D at 0x106efc0f0>]
```



Here we can see the asymptotic behavoir of the solutions of the ODE.

Systems of ODEs and the E-B Beam

Take our differential equation:

$$y''''(x) = \frac{f(x)}{IE}$$

We are simply going to turn this into a system of first order differential equations as such by allowing $y = U_0, y' = U_1, y'' = U_2, y''' = U_3$.

$$U_0 = U_1$$
$$U_1 = U_2$$
$$U_2 = U_3$$
$$U_3 = \frac{f(x)}{IE}$$

In [57]:

```
def I(w,d): return (w*d)/12
w = 0.3
d = 0.03
E = 1.3*(10**(10))
g = 9.81
def f(x): return -480*w*d*g
init_beam_cant = [0,0,f(0)/(12*I(w,d)*E),-f(0)/(12*I(w,d)*E)]
```

In [58]:

```
##### Here is our system of functions in python notation to be loaded into the solve
def ydot(t_i,y,i):
    z = np.zeros(4)
    z[0] = y[1]
    z[1] = y[2]
    z[2] = y[3]
    z[3] = f(i)/(I(w,d)*E)
    return z
```

In [59]:

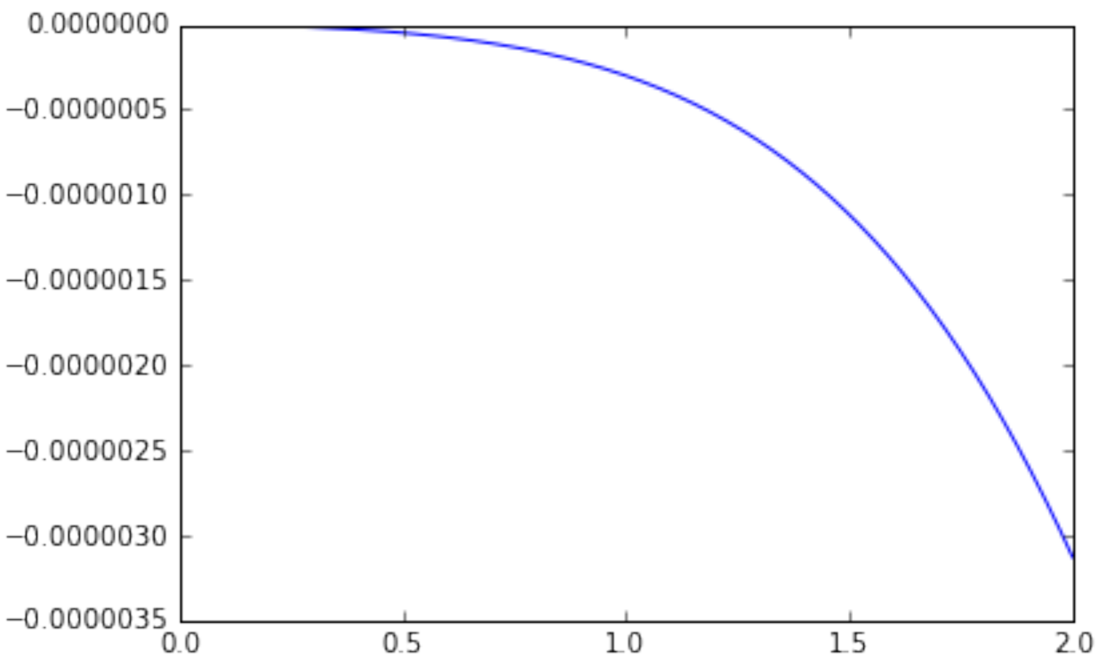
```
val,t = Runge_Kutta_vector(ydot,init_beam_cant,0,2,1000,4)
```

In [60]:

```
plt.plot(np.linspace(0,2,1000),val[:,0])
```

Out[60]:

```
[<matplotlib.lines.Line2D at 0x106f61400>]
```

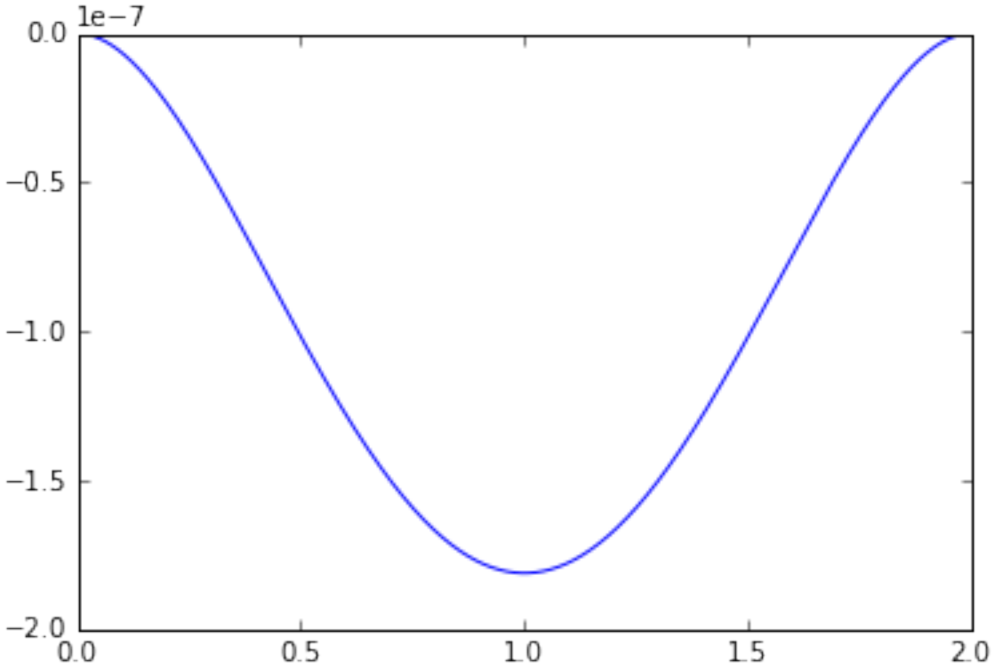


In [61]:

```
init_beam = [0,0,f(0)/(3*I(w,d)*E),-f(0)/(I(w,d)*E)]
val,t = Runge_Kutta_vector(ydot,init_beam,0,2,1000,4)
plt.plot(np.linspace(0,2,1000),val[:,0])
```

Out[61]:

```
[<matplotlib.lines.Line2D at 0x106f91278>]
```



In [ ]:

In [ ]: