Udai Sharma

PS-530

Computational Physics

23 February, 2023

Non Linear Harmonic Oscillator

The aim of the assignment is majorly divided into 2 segments.

Part I: The linear simple harmonic motion solver is extended to a non-linear harmonic oscillator with the introduction of the non linear correction. The solver is then verified using the method of manufactured solutions whereby a manufactured term is introduced to the nonlinear harmonic oscillator's equation of motion. The computer code is then tested whether it converges towards the manufactured solution at the expected rate. It is deemed to be verified if it does. After the code is verified, we turn off the manufactured solution knowing that if the solver is correct with the manufactured term, it will also be correct with the manufactured term omitted.

Part II: Once we have a verified solver for the nonlinear oscillator, the next step is to verify our calculation and check for any anomalies. In this case, The aim is to verify if the solver provides a correct solution under different conditions, specifically by changing our nonlinear constant. The effects of the nonlinear constant are investigated and a surprising loss of accuracy is observed. Finally, the aim is to understand the cause for the loss in this particular case and provide an appropriate reasoning for it.

WORKING

PART - I

1.1 Introducing the modified force law and the manufactured solution into the solver

- Setting the value of the non-linear oscillator constant ‘ **H = 0.4** ‘

```
k      = 1.0 # Spring constant
m      = 1.0 # Mass
cycles = 2.0 # No. of periods to integrate over
x0     = 1.0 # Initial displacement
v0     = 0.0 # Initial velocity
H      = 0.4 # Non-linear constant
```

- Modifying the initial conditions: The initial value of velocity ‘ ***v[0]*** ‘ is updated taking into account the non-linear correction ‘ ***1+H*x0*** ‘ and the manufactured source term ‘ ***H*(omega*x0*cos(omega*t0))^2*** ’

```
    t[0]  = 0.0
    x[0]  = x0
    v[0]  = v0 + 0.5*delta*( (1+H*x[0])*x[0]*omega**2 - H*(omega*x[0]*numpy.cos(omega*t[0]))**2)
```

- Modifying the value of velocity of the particle at the current time step ‘ ***i+1*** ‘ using the velocity at the previous step ‘ ***i*** ‘. The non-linear correction and the manufactured source term are taken into account for the modification.

```
    for i in range(steps):
        t[i+1] = t[i] + delta
        v[i+1] = v[i] + delta*( - (1+H*x[i])*x[i]*omega**2 + H*numpy.square(omega*x0*numpy.cos(omega*t[i])))
        x[i+1] = x[i] + delta*v[i+1]
    return t, x, v
```

- Defining the manufactured solution as the exact analytical solution ‘ ***x0*cos(omega*t)*** ‘ to compare with the numerical solution obtained using the Leapfrog method. It is used to verify the convergence of the numerical method using the ‘ ***L2 error norm*** ‘ which is the measure of relative error between the 2 solutions.

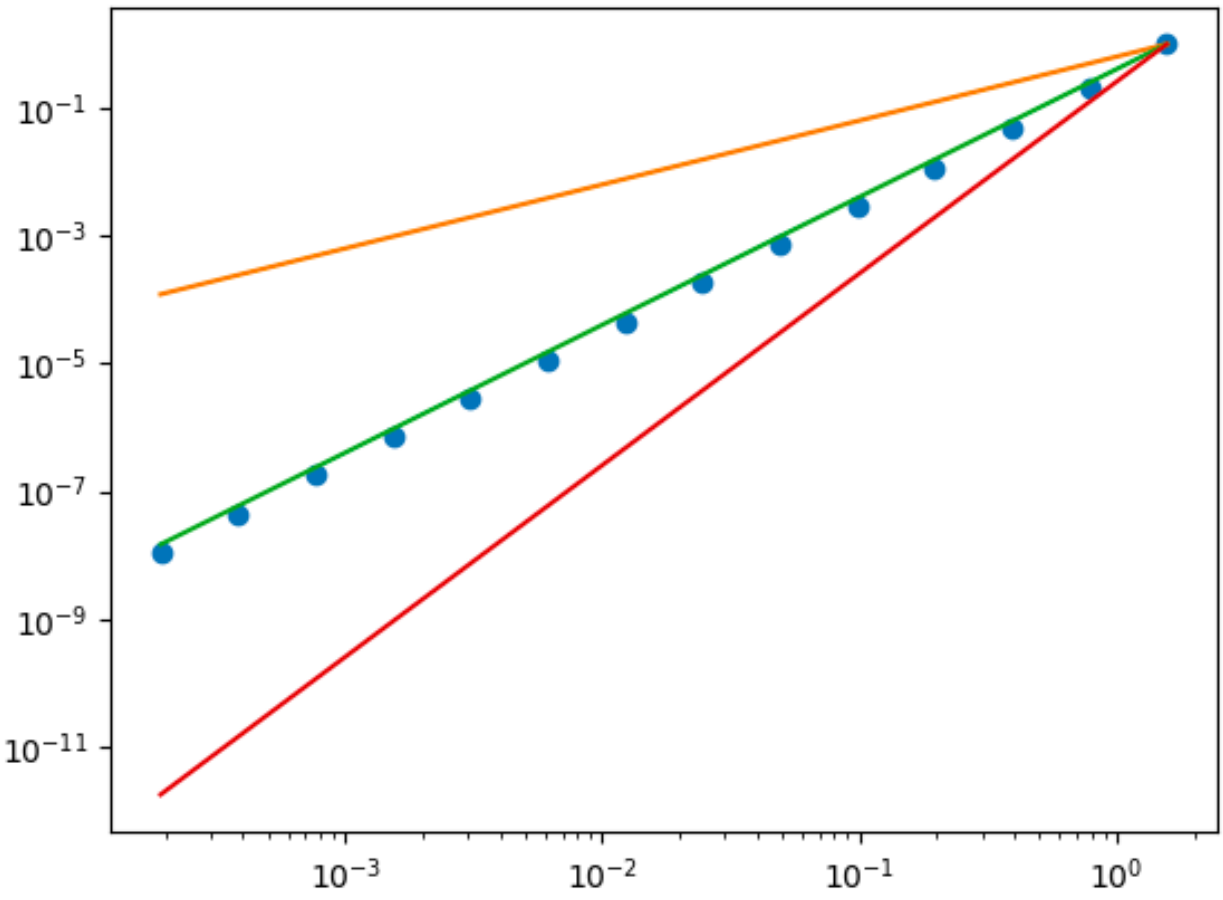
```
# Defining the Manufactured Solution
def manufactured_sol(t):
    omega = (k/m)**0.5
    return x0*numpy.cos(omega*t)
```

- Further ***‘ x_exact = manufactured_sol(t[i]) ‘*** is used to assign the pre-known analytical solution i.e. the manufactured solution to ***‘ x_exact ‘***. It is used twice in the code, once to calculate the ***‘ L2 error norm’*** and the other for plotting the analytical solution together with the numerical solution.

```
def l2_error_norm( t , x ):
    """Calculate the L2 relative error norm."""
    steps  = len( x ) - 1
    omega  = (k/m)**0.5
    l2_err = 0.0
    l2     = 0.0
    for i in range(steps):
        x_exact = manufactured_sol(t[i])
        l2_err += (x[i] - x_exact)**2.0
        l2     += x[i]**2.0
    return (l2_err/l2)**0.5


# The backend choice here may be platform dependent. You may need to
# change 'TkAgg' to something else.

plt.switch_backend( 'TkAgg' )

# This loop integrates the SHM equations repeatedly using an increasing
# number of steps (doubling at each loop iteration).
n        = 14
steps    = 8
l2_error = numpy.empty( n )
delta    = numpy.empty( n )
for i in range(0,n):
    t, x, v     = leapfrog( steps )
    x_exact     = manufactured_sol(t)
    delta[i]    = (k/m)**0.5*(t[1]-t[0])
    l2_error[i] = l2_error_norm( t , x )
    plt.plot( t , x )
    #plt.show( block=False )
    steps *= 2
```

1.2 Establishing that the manufactured solution is reproduced and that the rate of convergence is as expected.

- The ***‘ L2 relative error norm ‘*** is plotted against the time step ***‘ delta ‘*** : The graph showcases the convergence rate obtained from the leapfrog method. The graph shows 2nd order accuracy i.e. the convergence rate is as expected. Further, we can conclude that the code for the non-linear harmonic oscillator is verified.

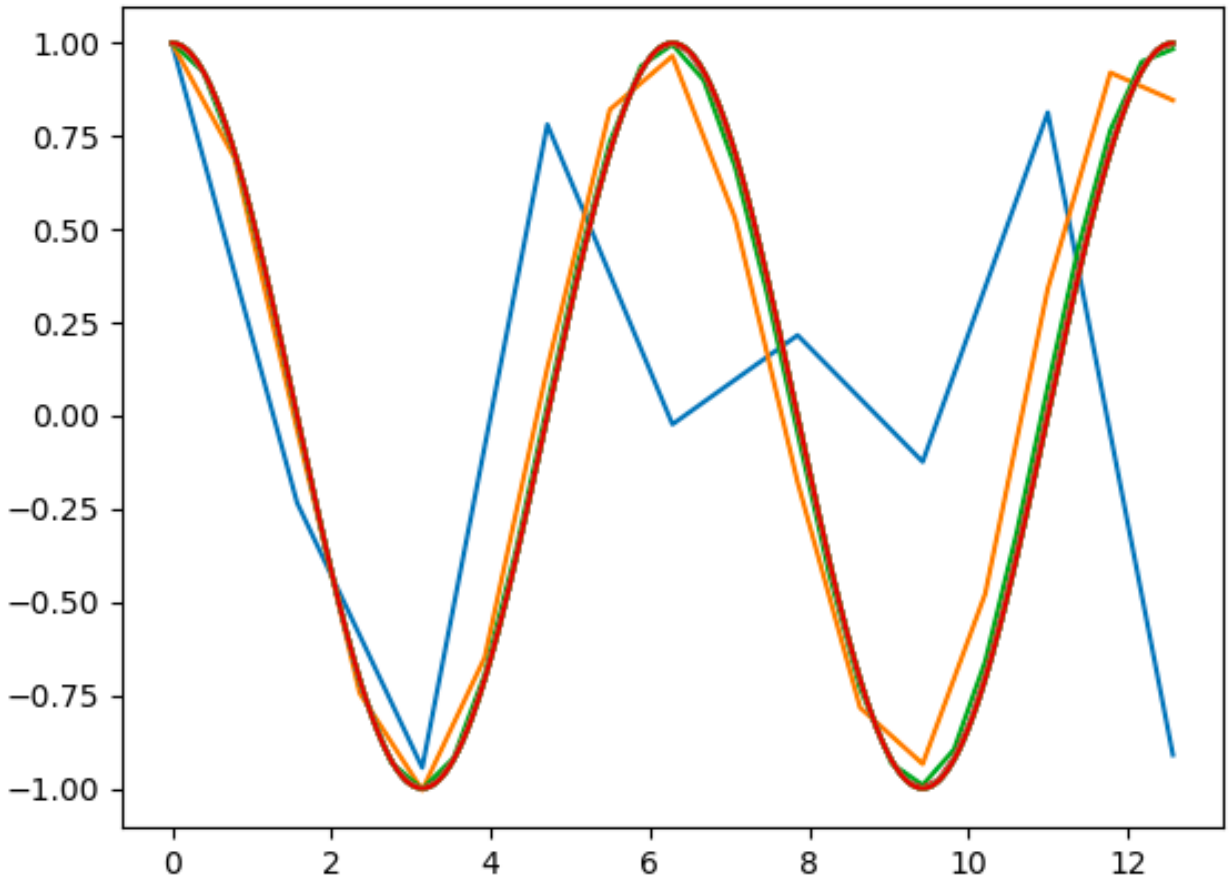


L2 Relative Error Norm vs Delta

- The position ***‘ x ‘*** is plotted against time ***‘ t ‘*** for different iterations with the number of steps doubling at each iteration. From the graph we can observe the dependence of convergence rate on the number of steps used in the leap frog function. As the number of steps increases with each iteration, the step size refines resulting in a finer time grid. This is visible in the following:

‘ delta = 2.0*cycles*numpy.pi/omega/steps ‘

This results in a higher convergence rate to the manufactured solution. Thus, we can conclude that the manufactured solution is reproduced.



Position Vs Time

1.3 Turning off the Manufactured solution

- The manufactured source term and the manufactured solution is removed from the initial conditions and the i iteration conditions as well to give us the formulas with just the non-linear correction.

```
import matplotlib.pyplot as plt
import numpy

k = 1.0  # Spring constant
m = 1.0  # Mass
cycles = 2.0  # No. of periods to integrate over
x0 = 1.0  # Initial displacement
v0 = 0.0  # Initial velocity
H = 0.4  # Non-linear constant

def leapfrog(steps):
    """Solve the simple harmonic motion equations for several oscillation cycles,
       assuming that the mass (m) and spring constant (k) are defined in the
       global space.
    """
    omega = (k / m) ** 0.5
    delta = 2.0 * cycles * numpy.pi / omega / steps
    x = numpy.empty(steps + 1)
    v = numpy.empty(steps + 1)
    t = numpy.empty(steps + 1)
    t[0] = 0.0
    x[0] = x0
    v[0] = v0 + 0.5 * delta * ((1 + H * x[0]) * x[0] * omega ** 2)
    for i in range(steps):
        t[i + 1] = t[i] + delta
        v[i + 1] = v[i] - delta * ((1 + H * x[i]) * x[i] * omega ** 2)
        x[i + 1] = x[i] + delta * v[i + 1]
    return t, x, v
```

Now that the manufactured solution is reproduced and the code is deemed to be verified. We can now turn off the manufactured source term and the manufactured solution. The solver we have now can be used to solve nonlinear problems but we know that a correct code does not guarantee correct solutions. So, considering this fact we move on to the next part of the assignment where we will try to verify our calculations and investigate why a surprising loss of accuracy occurs under some particular conditions.

PART-II

Now that we have turned off the manufactured solution, we will try to investigate the loss of accuracy that occurs on increasing the value of our nonlinear constant H. The code is modified to introduce a plot of the error in x against time i.e. **‘ x1 - x2 vs t ‘**. Where x1 and x2 are the values of position for different step sizes. Then, the change in this plot will be analyzed by changing the parameter H. Furthermore, I would also like to point out that the L2 error norm function has been removed with the intention of a clean code as it serves no purpose in this part of the assignment.

2.1 Surprising Loss of Accuracy on the increment of non-linear constant

First, the code is modified to investigate the error in position for two different step sizes with the non-linear constant set at 0.4. Afterwards, the results are plotted while varying the value of the non linear constant **‘ H ‘** from **0.4** to **0.5**.

- The loop for integrating SHM equations is modified. Where,

  - **‘ n ‘** is the number of iterations
  - **‘ steps1 ‘** and **‘ steps2 ‘** are the initial number of steps used in the leapfrog method for the two sets.
  - **‘ t1, x1, v1 ‘** and **‘ t2, x2, v2 ‘** are introduced in the leapfrog function corresponding to the number of steps **‘ steps1 ‘** and **‘ steps2 ‘**

```
n = 1
steps1 = 30
steps2 = 60
delta1 = numpy.empty(n)
delta2 = numpy.empty(n)
for i in range(0, n):
    t1, x1, v1 = leapfrog(steps1)
    t2, x2, v2 = leapfrog(steps2)
    delta1[i] = (k / m) ** 0.5 * (t1[1] - t1[0])
    delta2[i] = (k / m) ** 0.5 * (t2[1] - t2[0])
    steps1 *= 2
    steps2 *= 2
```

- ‘ **numpy.linspace** ‘ function is used to initialize a common time array to compare the results obtained from the two different step sizes. The function takes 3 arguments: the start time ‘ **0** ‘ , the end time ‘ **t1[-1]** ‘ and the number of points ‘ **len(t1)** ‘ and creates an array from ‘ **0** ‘ to ‘ **t1[-1]** ‘ with the same number of points as t1.

```
# Create a common time array using linspace.
t_common = numpy.linspace(0, t1[-1], len(t1))
```

- ‘ **numpy.interp** ‘ function is used to interpolate the solutions ‘ **x1** ‘ and ‘ **x2** ‘ on the common time array **‘t_common’**. The function takes 3 arguments: the new time array ‘ **t_common** ‘ , the initial time array **‘t1 or t2** ‘ and the initial position array ‘ **x1 or x2** ‘. The function returns an array of values at the new time points, so we finally obtain ‘ **x1_common** ‘ and ‘ **x2_common** ‘ with the same number of points as ‘ **t_common** ‘

```
# Interpolate x1 and x2 onto the common time array.
x1_common = numpy.interp(t_common, t1, x1)
x2_common = numpy.interp(t_common, t2, x2)
```

- Plotting: The following is the code which plots all the required graphs to understand the effects of varying the non linear constant.

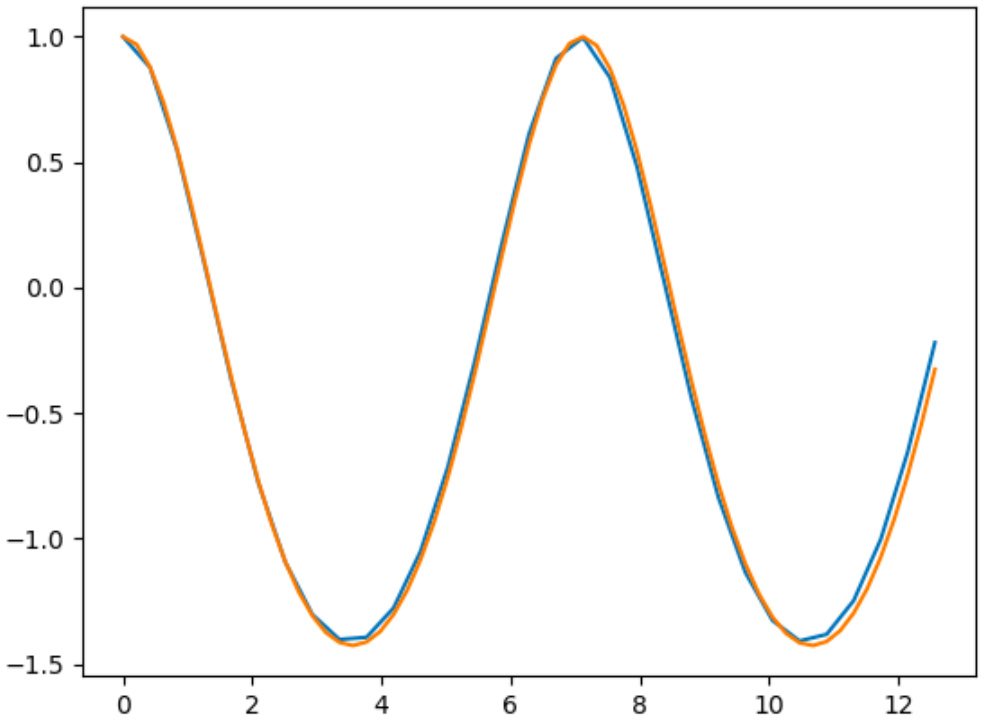
```
# Plot the difference between x1_common and x2_common against the common time.
plt.plot(t1, x1)
plt.plot(t2, x2)
plt.figure()
plt.plot(x1, v1)
plt.figure()
plt.plot(x2, v2)
plt.figure()
plt.plot(t_common, x1_common - x2_common)
plt.xlabel('Time')
plt.ylabel('x1 - x2')
plt.show()
```

- **‘ x1 vs t1 ‘** and **‘ x2 vs t2 ‘** are plotted in the same plot window to visualize the difference in the position for two different step sizes.

For **H = 0.4**,

The blue line shows **‘ x1 vs t1 ‘** at which **‘ steps = 30 ‘**

The red line shows **‘ x2 vs t2 ‘** at which **‘ steps = 60 ‘**

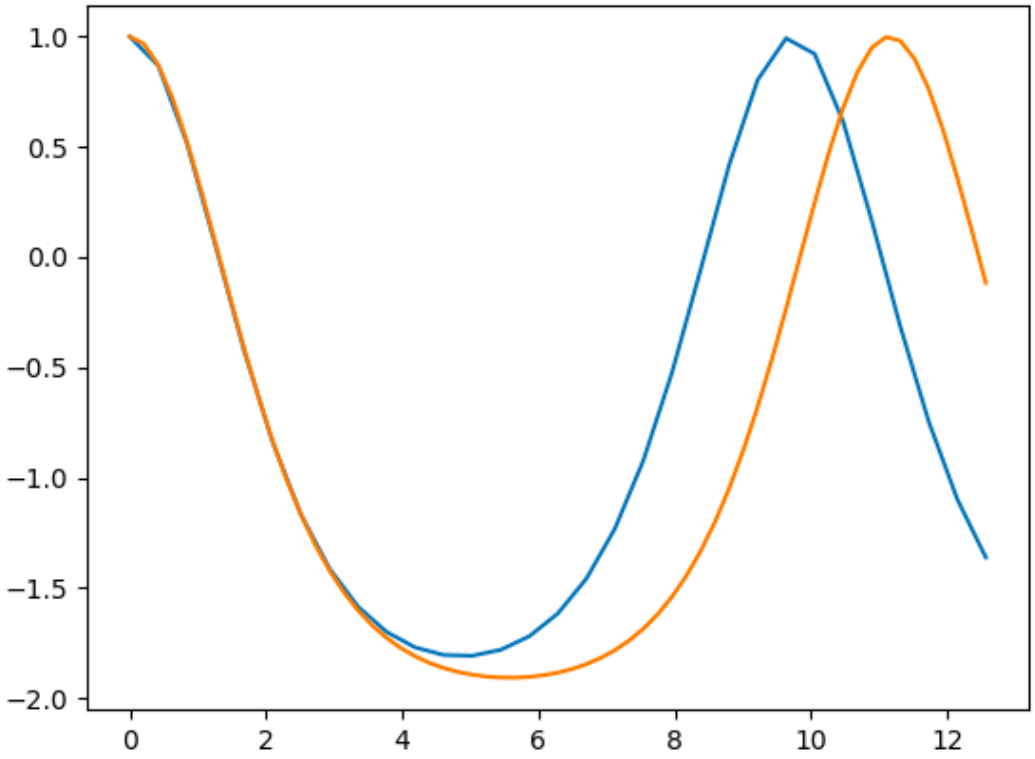


xi Vs ti (i=1,2) for H = 0.4

For **H = 0.5**,

The blue line shows **‘ x1 vs t1 ‘** at which **‘ steps = 30 ‘**

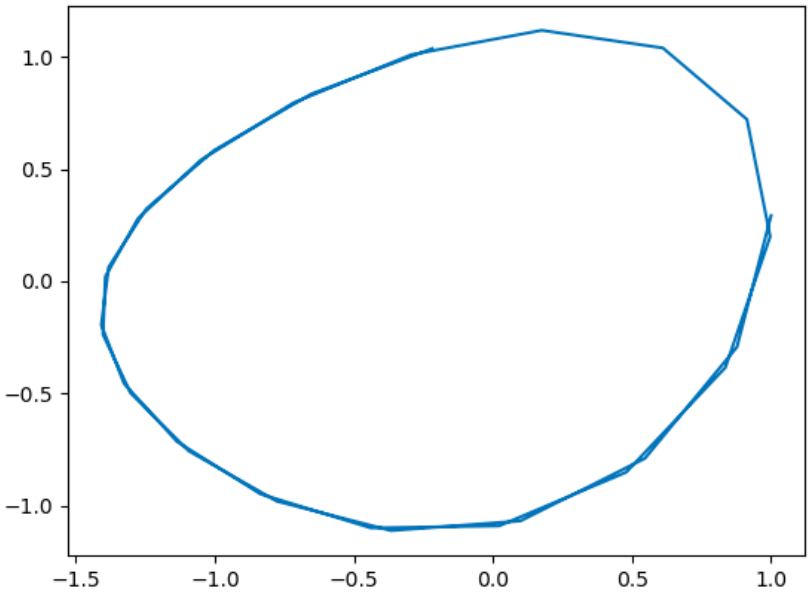
The red line shows **‘ x2 vs t2 ‘** at which **‘ steps = 60 ‘**



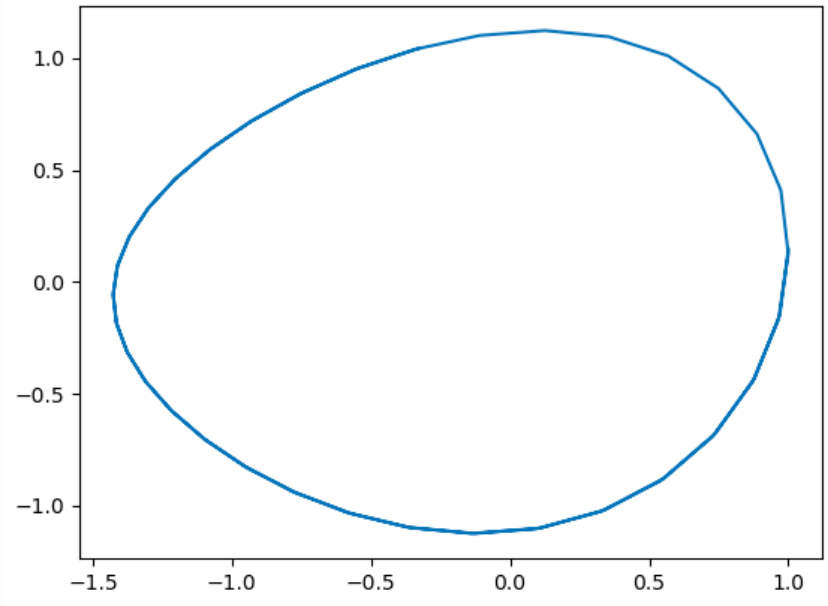
xi Vs ti (i=1,2) for H = 0.5

- Plot of **‘ x1 vs v1 ‘** and **‘ x2 vs v2 ‘** : If we plot x vs v, we obtain what is called a phase space plot. This plot shows the trajectory of the oscillator in the x-v plane, and it can give us insights into the behavior of the system. For example, we can see whether the oscillator is undergoing simple harmonic motion (circular trajectory), damped oscillations (spiraling inwards), or forced oscillations (elliptical trajectory). In the following graphs we can observe that the results have a bit of ellipticity in them which is due to the fact that our oscillator is not exactly a simple harmonic motion due to the non linear correction, and as we increase the value of H we see that the significance of the non-linear term increases and that is why the graphs with H = 0.5 are more error prone and sharply elliptical than the H = 0.4.

For **H = 0.4** ,

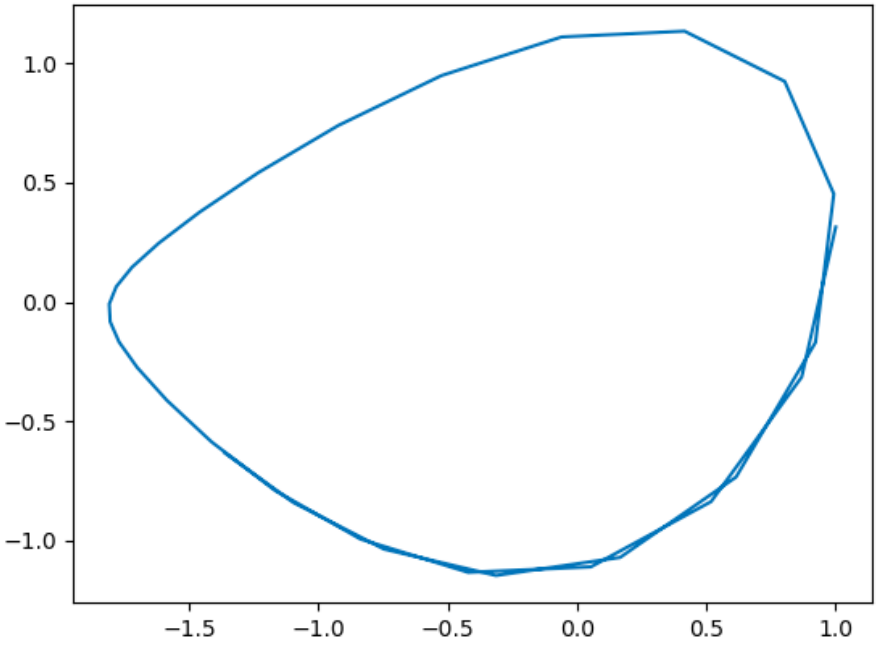


x1 vs v1

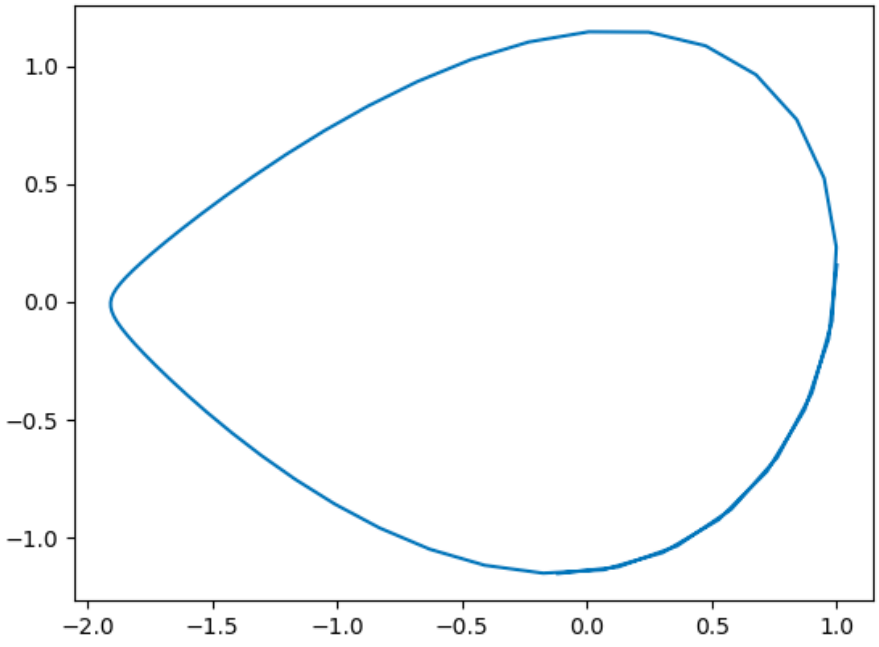


x2 vs v2

For **H = 0.5**,



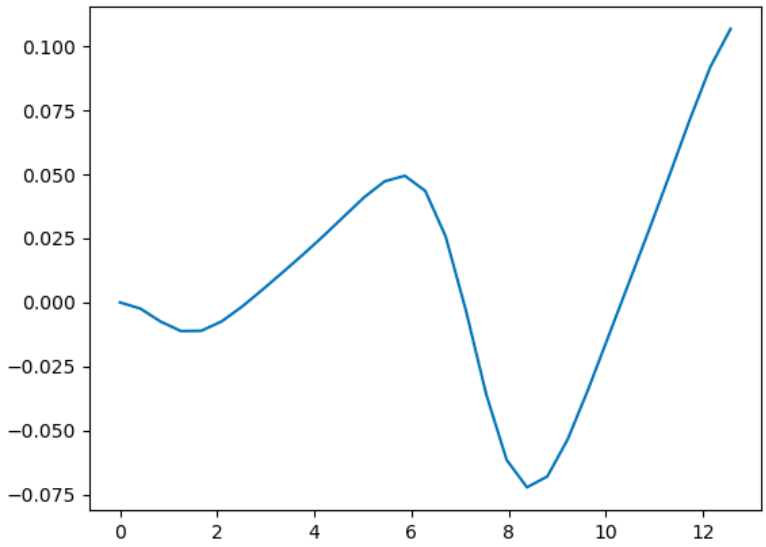
x1 vs v1



x2 vs v2

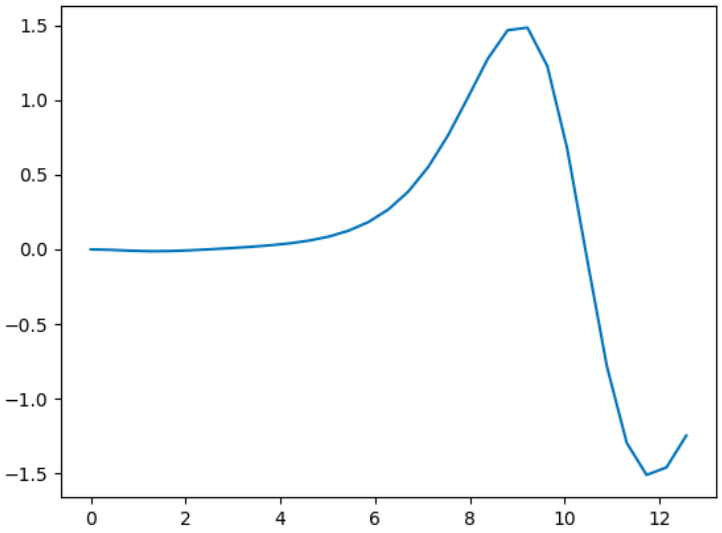
- ‘ **x1_common - x2_common ‘ vs ‘ t_common ‘** is plotted to visualize the magnitude of error in the position for two different step sizes.

For **H = 0.4**: The absolute error observed seems to be acceptable with the peak at 0.1and the plot is also qualitatively correct.



Absolute Error vs Time

For **H = 0.5**: The absolute error has increased significantly to 1.5. This is the unexpected loss of accuracy which is observed in our nonlinear oscillator.



Absolute Error vs Time

2.2 Why does the nonlinear oscillator show an error on increasing the value of H?

There can be various effects that can lead to a significant increase in error on increasing the value from H but the most significant error that can be pointed out is that the system starts to become highly non linear due to the term ‘ **H*x^2** ‘ in ‘ x*(1+H*x) ‘. The motion is assumed to be roughly harmonic in the numerical solution of the SHM system, which makes use of approximations. The approximations lose accuracy as the system's nonlinearity grows, which causes bigger mistakes in the numerical solution. The assumption of small amplitude oscillations, which is used to construct the equations of motion for SHM, is broken, in particular when H is large. As a result, the system's nonlinearity causes stronger oscillations. When H rises, the numerical solution consequently becomes less precise, resulting in increased errors.

However, when H is big, the system may exhibit complex nonlinear phenomena that are problematic to accurately model using numerical techniques. These nonlinear tendencies have the potential to cause numerical solution instability, which in turn has the potential to produce greater errors.

Therefore, It is crucial to select an appropriate numerical approach that can manage the system's nonlinearities and yield accurate results for higher levels of H. This is the reason why error analysis and convergence tests need to be carried out to make sure the numerical solution is trustworthy and correct.