# 7 Sep 23 - Activity: Numerical Integration and More Lagrangians

Now that we have an idea how to find a wealth of interesting ordinary differential equations using Lagrangian mechanics, we’ll work on building up ways to understand these equations, their solutions and behavior. The issue with this is that **most ODEs do not have analytical solutions**. That means we can’t write down nice closed-form solutions for them using transcendental functions. However, don’t despair, because that does not mean there is no solution. In fact, the vast majority of non-pathological ODEs one might come across in physics are **guaranteed** to have unique solutions (at least for finite time). We can easily calculate these solutions using **numerical integration**. Next week we’ll also see how we can characterize the behavior of ODEs even without access to numerical integration.

## The Simple Harmonic Oscillator

Let’s start simple with everyone’s favorite differential equation, the simple harmonic oscillator. Recall that we can write the SHO as:

where . This equation is 2nd order, but numerical integration techniques only work on 1st order equations. Thankfully they work on any number of potentially coupled 1st order equations. This means that with a quick change of variables, we can write the SHO as a system of 2 first order equations by introducing a new variable equal to the velocity of the oscillator.

Then the acceleration of the oscillator can be written as:

This trick for writing higher order differential equations as first order equations is incredibly common.

### Setting up to numerically integrate

We need a few things to numerically integrate using solve\_ivp in python. First, we import the relevant libraries and functions.

# Import libraries  
import matplotlib.pyplot as plt  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy.integrate import solve\_ivp

\*\*Note: The code in this block is not meant to be run directly, it is just a reference for the steps we need to take. Working code that is put together appears in the next cell.\*

#### 1. Derivatives Function

First, we need to set up a derivatives function that calculates and returns a list of the values of the first order derivatives given an input list of current values. These current values represent a location in **phase space**. As we will learn, [phase space](https://en.wikipedia.org/wiki/Phase_space) is a space that contains all the information about the state of an ODE. The simple harmonic oscillator has a 2D phase space since its state is totally defined by its position and velocity.

Here’s what our derivatives function looks like for a SHO:

def diffyqs(t, curr\_vals, omega2):  
 # 2 first-order differential equations for a SHO  
 # first 2 arguments are always t and curr\_vals, which are followed by any parameters of your ODEs  
 x, v = curr\_vals # unpack current values  
   
 vdot = -omega2 \* x # calculate derivative  
  
 return v, vdot # return derivatives

We will pass this function to our solver, which will give us back integrated solutions of our list of derivatives. So since , our solution will return first, and .

#### 2. Time Setup

We need to define the time span to solve the ODE for AND the specific times we’d like solution points for. Here it is also convienient to choose a time step . Here’s one way we could do this in python:

tmax = 15  
dt = 0.1  
tspan = (0, tmax) # time span  
t = np.arange(0, tmax, dt) # specific times to return solutions for

#### 3. Parameters and Initial Conditions

Since we’re dealing with ODEs, we need to supply an initial condition to be able to solve. The SHO has 2D phase space so we need 2 values for our initial condition. We’ll also define parameter value(s) in this step.

omega2 = 2  
initial\_condition = [1, 0] # pull back 1m, no initial velocity

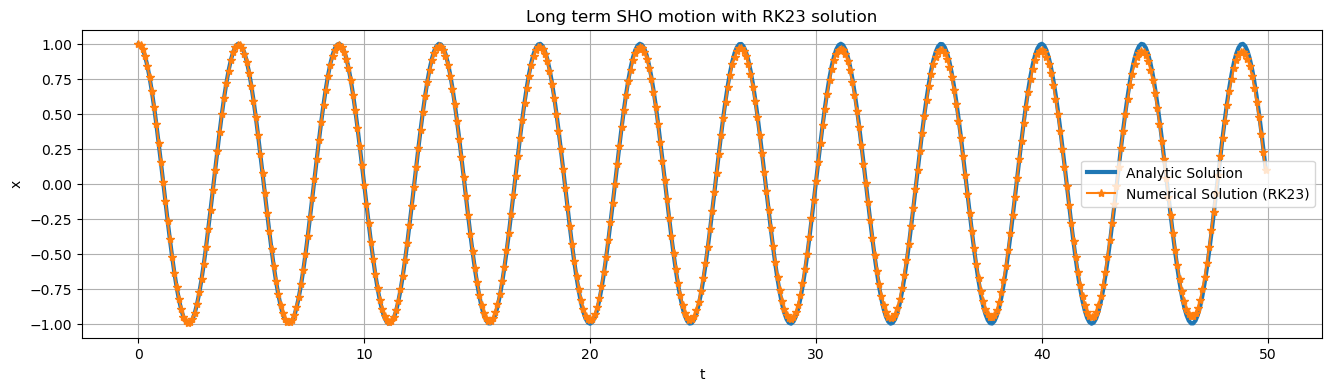
#### 4. Call Integrator

Now all we have left to do is to actually use solve\_ivp to do the integration. The syntax for how to do this is shown below. We also get the opportunity to tell solve\_ivp exactly what numerical integration method we’d like it to use. For now we can think of the integrator as a magic box and choose RK45, or a Runge-Kutta 4th order method.

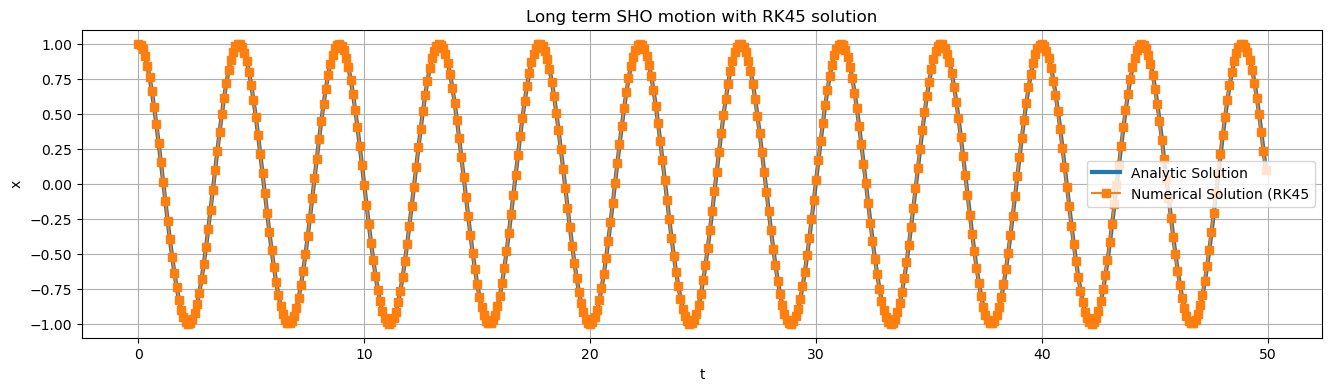
solved = solve\_ivp(diffyqs, tspan, initial\_condition, t\_eval = t, args = (omega2, ), method="RK45")

To access the solution directly, use solved.y. The variable solved.y[0] is the solved for position array and solved.y[1] is the velocity array in this case. Now let’s see a full implementation of this below, including some visualization that compares our numerical solution to the analytical solution of the SHO.

# 1. Derivatives Function  
def diffyqs(t, curr\_vals, omega2):  
 x, v = curr\_vals   
 vdot = -omega2 \* x  
 return v, vdot  
  
# 2. Time Setup  
tmax = 50  
dt = 0.1  
tspan = (0, tmax)  
t = np.arange(0, tmax, dt)  
  
# 3. Parameters and Initial Conditions  
omega2 = 2  
initial\_condition = [1, 0]   
  
# 4. Call Integrator (note we can swamp them out, RK45 is the default)  
RK23solved = solve\_ivp(diffyqs, tspan, initial\_condition, t\_eval = t, args = (omega2,), method="RK23")  
RK45solved = solve\_ivp(diffyqs, tspan, initial\_condition, t\_eval = t, args = (omega2,), method="RK45")  
  
# 5. Visualization and Comparison to analytical solution  
def analytic\_sol(t, omega0, initial\_condition):  
 x0, v0 = initial\_condition  
 return (v0/omega0)\*np.sin(omega0\*t) + x0 \* np.cos(omega0\*t)  
  
plt.figure(figsize=(16, 4))  
plt.plot(t, analytic\_sol(t, omega2\*\*0.5,initial\_condition), label = "Analytic Solution", linewidth = 3)  
plt.plot(t, RK23solved.y[0], label = "Numerical Solution (RK23)", marker='\*')  
plt.title("Long term SHO motion with RK23 solution")  
plt.xlabel("t")  
plt.ylabel("x")  
plt.legend()  
plt.grid()  
  
plt.figure(figsize=(16,4))  
plt.plot(t,analytic\_sol(t, omega2\*\*0.5,initial\_condition), label = "Analytic Solution", linewidth = 3)  
plt.plot(t,RK45solved.y[0], label = "Numerical Solution (RK45", marker='s')  
plt.title("Long term SHO motion with RK45 solution")  
plt.xlabel("t")  
plt.ylabel("x")  
plt.legend()  
plt.grid()



png



png

## Group Activity

### Back to Paraboloid Paradise

[Analytical Solution (Finding ODEs for Paraboloid)](../../assets/notes/Notes-Bead_in_a_Paraboloid.pdf)

Let’s consider the problem where a particle was constrained to move on the surface . The EOM we arrived at are complex

**✅ Do this**

Introduce variables and to use our trick for reducing order differential equations to first order equations to write the equations of motion for this problem as a system of four first order differential equations (shown below). **You may set c=1, but keep it as a variable.**

**✅ Do this**

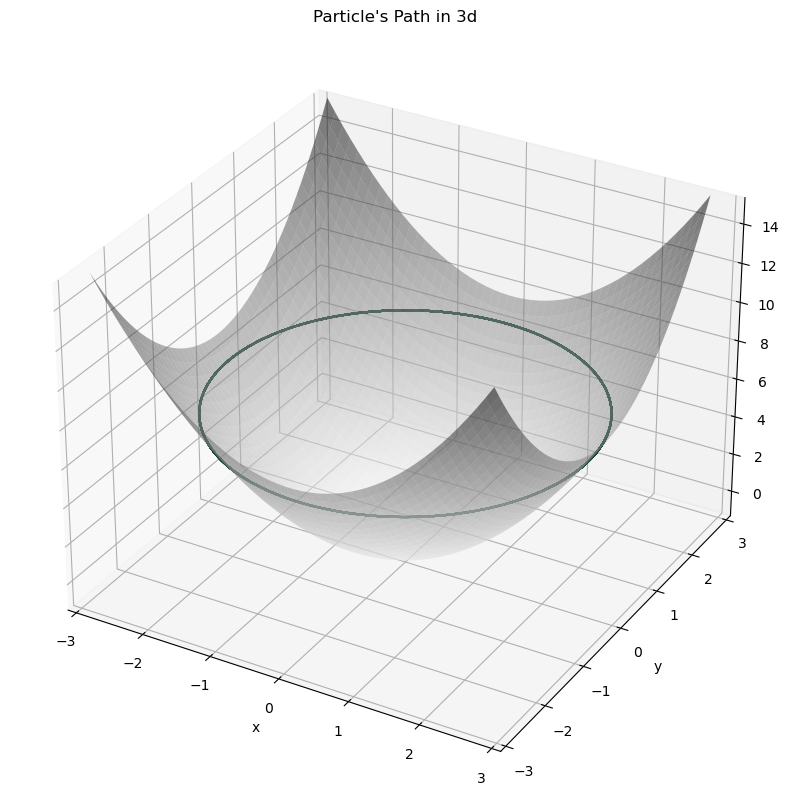
Use these equations to correct the diffyqs function in the cell below.

# 1. Derivatives Function  
def diffyqs(t, curr\_vals, g, c):  
  
 r, v, theta, omega = curr\_vals  
   
 vdot = 0  
  
 omegadot = 0  
  
 return v, vdot, omega, omegadot # solution will return in this order, but integrated (r,v,theta,ω)  
  
# 2. Time Setup  
tmax = 40  
dt = 0.01 # unneccecarily small dt to make plot super smooth  
t = np.arange(0, tmax, dt)  
  
# 3. Parameters and Initial Conditions  
c = 1  
g = 9.81  
x0 = [2.6,0,0,2]   
  
# 4. Call Integrator  
solved = solve\_ivp(diffyqs, (0, tmax), x0, t\_eval = t, args = (g, c, ), method="RK45")

**✅ Do this**

1. Make r vs t and theta vs t plots of this trajectory. Can you think of what that trajectory would look like in cartesian coordinates? 2. Run the cell below to see what the trajectory looks like in 3D. How does the true trajectory compare to your prediction?
2. Change the initial condtion to examine the following cases and plot the trajectories in 3d:
   1. Particle starts from rest and is let go
   2. Particle starts at a given height and is given a low speed (less than needed to orbit)
   3. Particle starts at a given height and is given a low speed (more than needed to orbit)
   4. Can you find a flat horizontal circular orbit?

def parabaloid(x, y, alpha=1.):  
 # function of a paraboloid in Cartesian coordinates  
 return alpha \* (x\*\*2 + y\*\*2)  
  
def cylindrical\_to\_cartesian(r, th, alpha=1.):  
 # convert back to cartesian coordinates for ease of plotting  
 r = np.array(r)  
 th = np.array(th)  
 x = r\*np.cos(th)  
 y = r\*np.sin(th)  
 return x,y,parabaloid(x, y, alpha)  
  
def plot\_solution(solved):  
 # Function to plot the trajectory   
  
 # points of the surface to plot  
 x = np.linspace(-2.8, 2.8, 50)  
 y = np.linspace(-2.8, 2.8, 50)  
 alpha = c  
 # construct meshgrid for plotting  
 X, Y = np.meshgrid(x, y)  
 Z = parabaloid(X, Y, alpha)  
  
 # get trajectory in cartesian coords  
 xtraj, ytraj, ztraj = cylindrical\_to\_cartesian(solved.y[0], solved.y[2], alpha)  
  
 # plot plot plot  
 fig = plt.figure(figsize = (10,10))  
 ax = plt.axes(projection='3d')  
 plt.title("Particle's Path in 3d")  
 ax.plot\_surface(X, Y, Z, cmap='binary', alpha=0.5)   
 ax.plot3D(xtraj, ytraj, ztraj, c = "#18453B")  
 ax.set\_xlim(-3, 3); ax.set\_ylim(-3, 3); ax.set\_zlim(-1 ,15)  
 ax.set\_xlabel('x')  
 ax.set\_ylabel('y')  
 ax.set\_zlabel('z')  
 plt.show()  
  
plot\_solution(solved)



png