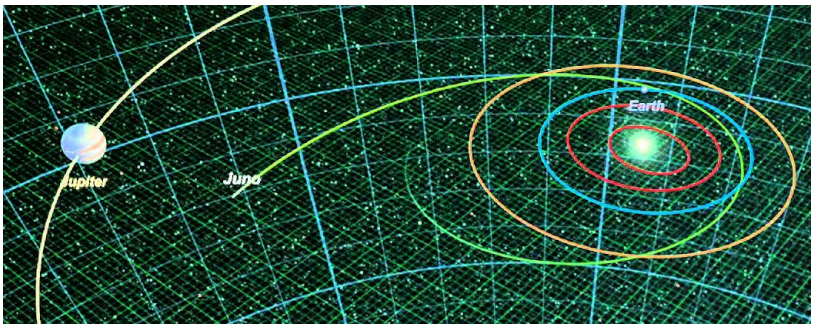
Python ODE Solvers
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/ode_python/ode_python.pdf

---



You don't get a second chance in planetary travel!

"Python ODE Solvers"

- *ODE solvers offered by Python can handle more difficult problems;*
- *The solver can determine the appropriate stepsize;*
- *The solver can estimate the error being made;*
- *The solver can vary the order of the method.*
- *The solver can handle "stiff" problems.*
- *The user can control the output points and other solution parameters.*

1 Why system solvers are needed

Any physical system that changes in a smooth way can be modeled by a set of ordinary differential equations. And there are many ways to write such a system, some of which are much harder to solve than others. In some cases, solving the ODE for a physical system is like walking along a very sharp mountain ridge. Even a tiny step away from the correct path will lead you to a completely different result. Calculations involving the weather are an example where even a small change in the initial condition is enough to utterly change the solution path.

Reducing the stepsize can often improve accuracy. Changing the formula used to estimate the solution at the next time step can also help. Estimating the error allows us to determine whether these changes are necessary. But all these adjustments come at a cost, and if done carelessly, it can be impossible to get an accurate solution efficiently. That is the job that we expect a system ODE solver to take over for us.

Python supplies ODE solvers that do an excellent job of handling systems that are much too hard for the average user to deal with. We will look at a few difficult problems and see how to get a good answer.

2 A stiff problem

Here's an example of an ODE whose solution is somewhat like walking along a mountain ridge. We can write it generically as:

$$y' = \lambda * (\cos(t) - y)$$
$$y(t_0) = y_0$$

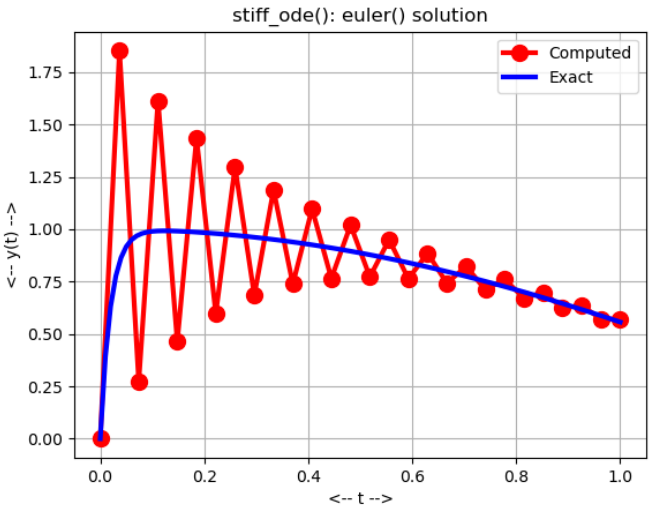
and we will pick the parameters to be

$$t_0 = 0.0$$
$$y_0 = 0.0$$
$$\lambda = 50.0$$
$$t_{max} = 1.0$$
$$n = 27$$

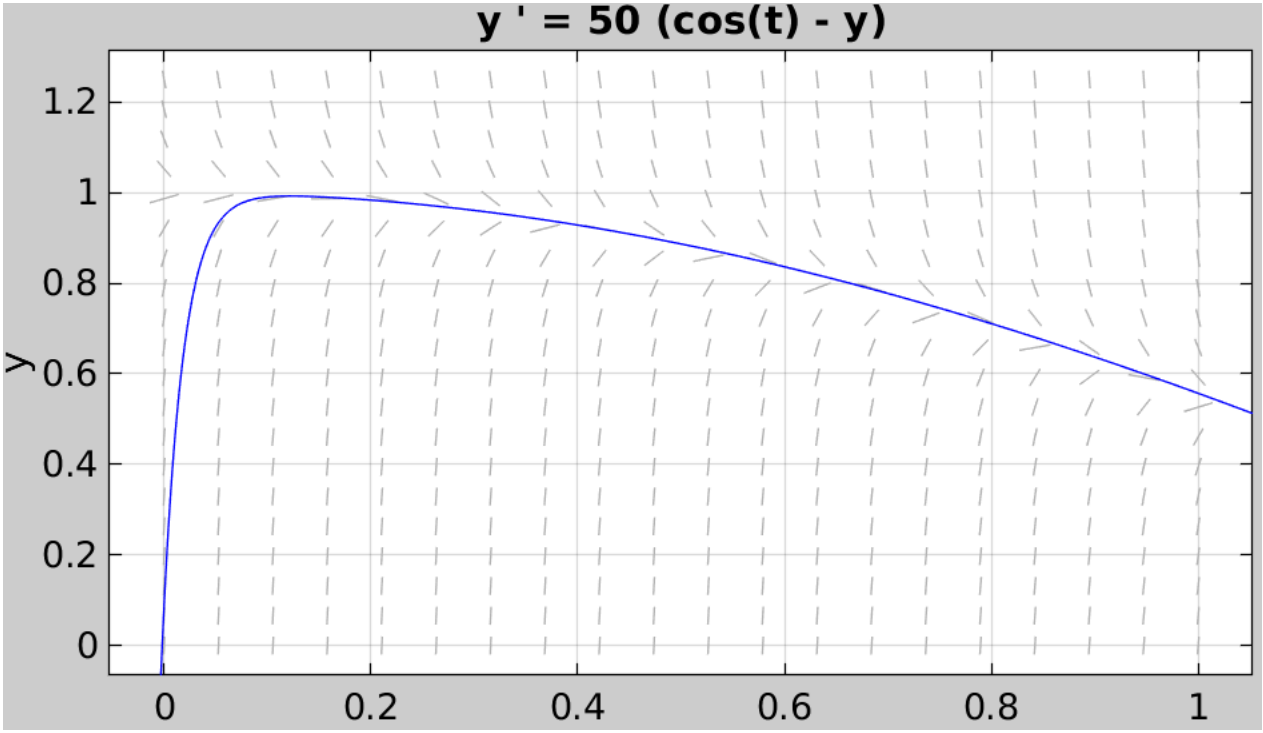
where n is the number of equal steps we plan to take over the interval $[0.0, 1.0]$. The exact solution of this problem is:

$$y(t) = \lambda \frac{\sin(t) + \lambda \cos(t) - \lambda e^{-\lambda t}}{\lambda^2 + 1.0}$$

If we compare the exact solution to what `euler_solve()` returns we see a surprisingly bad result:



We can almost understand why the very first step is so bad, but how do we explain this crazy, persistent oscillation?



Pick any value of time t, and look just above and just below the blue line that represents the exact solution. The direction field arrows are not close to being tangents to the curve. Instead, they are pointing very sharply up and down. This is because the entire solution field wants to get as close as possible to the blue curve, so every solution curve zooms up or down to the blue curve and only turns close to the tangent direction at the last possible moment. This means that an ODE solver that tries to get the proper direction has to be very close to the true solution in order to get useful information. If it is just a little away from the true solution, it is told to jump up or jump down. This is why the approximate solution is so jagged.

3 Introducing `solve_ivp()`

The `scipy` library includes a powerful general purpose ODE solver called `solve_ivp()`, that does a good job with equations like our stiff system. The format for calling this function is similar to the interface for `euler_system()`, leaving off n, the number of steps, and returning a single solution object `sol` that we have to "unwrap" to get our results.

Here is a typical usage:

```
from scipy.integrate import solve_ivp
import numpy as np

tspan = np.array ( [ 0.0, 1.0 ] )
y0 = np.array ( [ 0.0 ] )          # y0 must be an array, not a number!

sol = solve_ivp ( stiff_dydt, tspan, y0 )
```

A few things may still be mysterious:

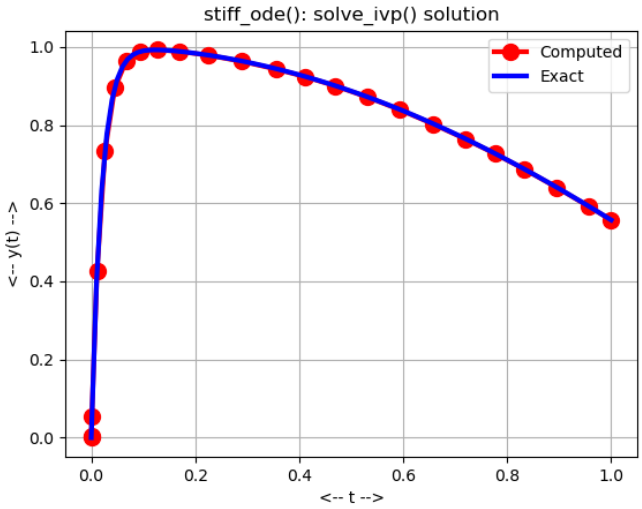
- The input quantity `y0` must be an array, even if we are only dealing with one equation. In this case, we just have to wrap our numeric value inside square brackets;

- The output quantity `sol` contains the solution information. `sol.t` is the array of times. `sol.y[0]` is the array of values of the first component If there are more components, they are in `sol.y[1]` and so on.

Now if we wish to plot the solution data, we use the `sol` structure:

```
plt.plot ( sol.t, sol.y[0], 'ro', markersize = 10 )
```

When we plot points along the solution, we see what seems close to perfect agreement with the exact solution. Note that the code took several small steps in time at the beginning, to get to the bend, after which the solution is smooth and easy to follow.



4 Retry the predator-prey problem

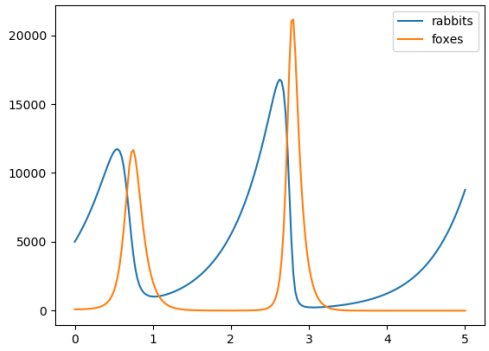
In the previous class, we used the Euler method to solve a pair of predator prey equations. The solution was rough, and did not show nice periodicity, presumably because we only used 200 steps. It was our task to decide how many steps `n` to specify, so that we got a sufficiently accurate answer. In contrast, `solve_ivp()` is able to estimate how well it is approximating the answer, and will vary the stepsize as needed.

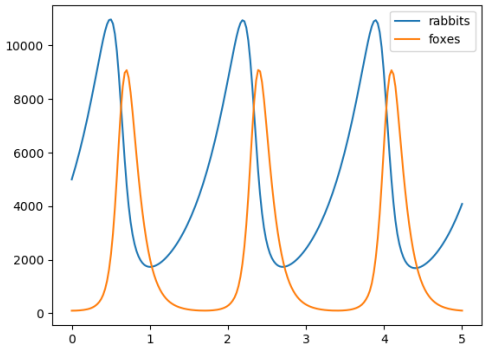
Using the call

```
sol = solve_ivp ( predator_prey_dydt, tspan, y0 )
```

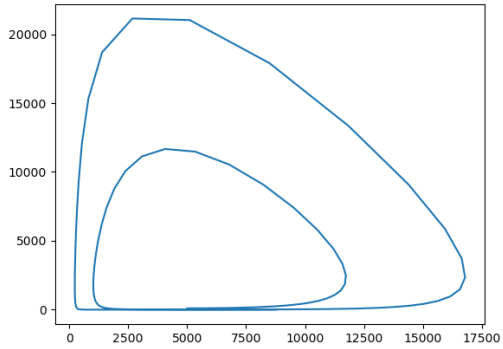
we can solve the predator prey problem again, plot the same quantities as before, and compare with the results using 200 steps of our Euler method.

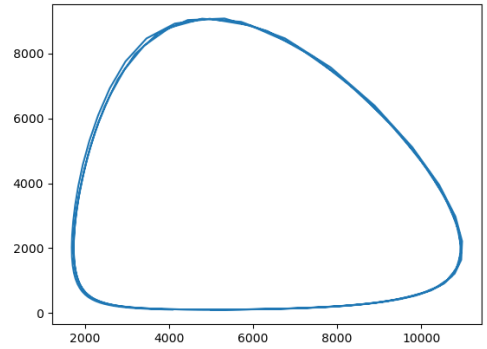
The time plot shows how inaccurate our Euler solution was. Instead of $2\frac{1}{3}$ peaks, we have 3 full peaks for both species. The three peaks look to have essentially the same shape and height.



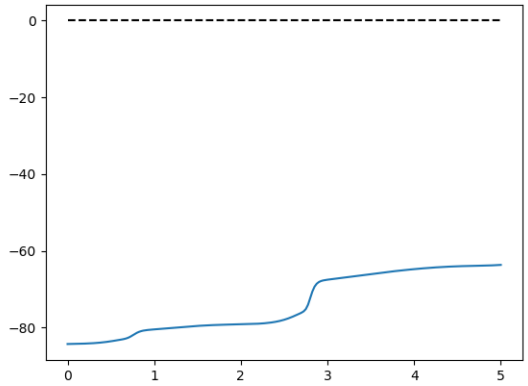


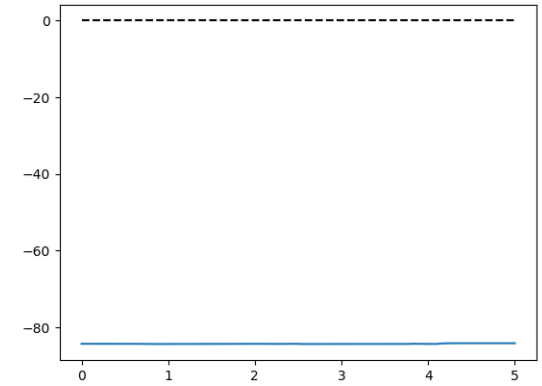
The curve in the phase plane plot has been traced over 3 times, but you can just barely see that there is some variation in the data.





The conservation measurement is flatlining, the way a perfect solution should.





5 How to use `solve_ivp()`

You get access by the command

```
from scipy.integrate import solve_ivp
```

As always, get help by the command `help ( solve_ivp )`.

The code measures its work by counting the number of times it calls the ODE right hand side function. This quantity is stored as `sol.nfev`. It is not the same as the number of time steps, which is `len(sol.t)` and is usually much smaller.

You can solve scalar ODE's, but you have to make sure that your initial condition is input as a vector. For instance, if the initial condition is 10, you pass [10.0], or a variable `y0=[10]`, not just the number 10, and not a variable just set to a number, `y0=10`.

The output quantity `sol` includes all the information about the solution. In particular, it returns a time vector `sol.t` and a solution array `sol.y`. If your ODE is a system, this `sol.y` data is returns as a "sideways" vector, not an "up and down" vector. In other words, the first component of the solution is stored as `sol.y[0]`. (which is like a row) rather than `sol.y[:,0]` (which would be like a column). If you need to work with column data, the easiest thing to do is make an up-and-down copy by

```
y = np.transpose ( sol.y )
```

The code can take big steps, which mean that plotting the default pairs of (t, y) or making a phase plane plot may result in a jagged and deceptive image. If you want to control where the results are reported, use the `t_eval=?` option:

```
t = np.linspace(t0,tmax,500)
sol = solve_ivp ( stiff_dydt, tspan, y0, t_eval = t )
```

and now the data in `sol.t` and `sol.y` will be at the finer mesh you requested.

By default, `solve_ivp()` uses the 'RK45' ODE solver. If the ODE is "stiff", then you can use the `method=?` argument to switch to another solver, presumably 'BDF' or 'RADAU.

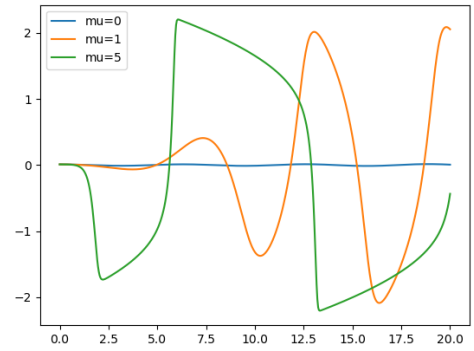
```
sol = solve_ivp ( stiff_dydt, tspan, y0, method = 'BDF' )
```

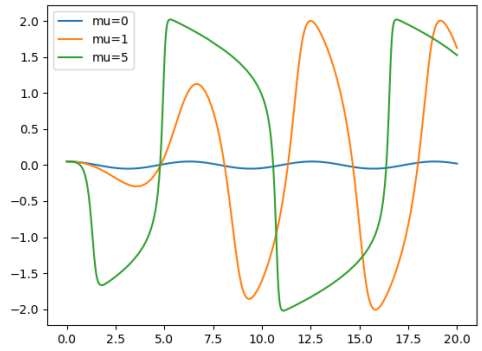
6 Retrying the van der Pol equation

Let's return to the van der Pol equation, and try our `solve_ivp()` ODE solver. I am going to change the initial conditions, because the value `y0=[1,0]` means that the solution starts out with the right amplitude. Setting a bigger value for u_0 means the solution will gradually decrease to its desired size, while a smaller initial value results in an initial stage of growth.

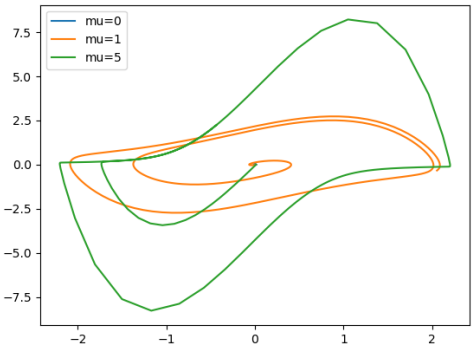
Having said that, let's compare solutions for $\mu = 0, 1, 5$ using `euler_system()` and `solve_ivp()`:

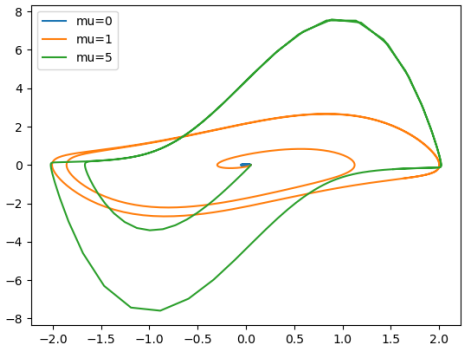
The time plots match somewhat, although there are clearly difference. We used 500 steps for each use of the Euler method, whereas `solve_ivp()` used 87, 193, and 289 steps respectively for the three values of μ. The time plots show that for our initial condition of `[0.25,0.0]`, the solution does go through a short period of growth before hitting periodicity. It also suggests that as μ increases, the period of the solution increases as well.





The phase plane plot suggests that after a very short time, the solution reaches the periodic behavior and stays there. We can check this by extending the time period further, by looking at other values of μ, and by varying the initial condition.





7 The Arenstorf orbit

In 1963, Richard Arenstorf worked out a set of equations for the stable periodic path of a satellite that intersects the moon's orbit. We let the earth be at the origin, and the moon's orbit is modeled as a circle of radius 1. Let (x, y) represent the location of the satellite at time t, and (x', y') be the velocity components. Then the satellite's path can be determined from the equations

$$x'' = x + 2y' - M_e \frac{x + M_m}{D_1} - M_m \frac{x - M_e}{D_2}$$
$$y'' = y - 2x' - M_e \frac{y}{D_1} - M_m \frac{y}{D_2}$$

where

$$D_1 = ((x + M_m)^2 + y^2)^{3/2}$$
$$D_2 = ((x - M_e)^2 + y^2)^{3/2}$$

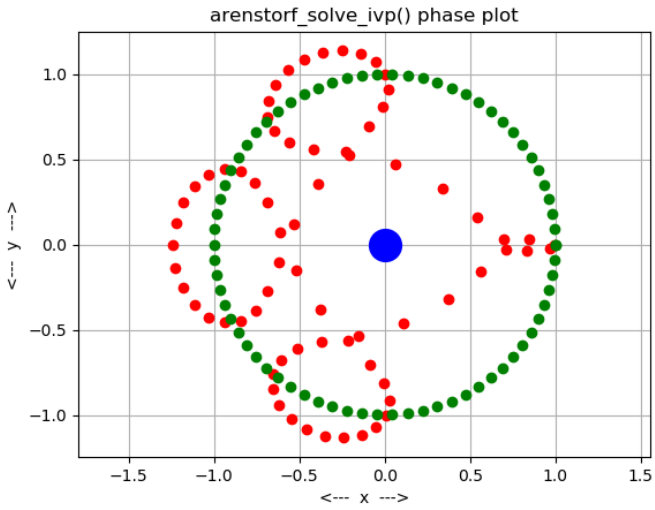
where M_e and M_m are the normalized masses of the Earth and the moon, so that

$$M_e = 0.987722529$$
$$M_m = 0.012277471$$

with initial conditions

$$x(0) = 0.994$$
$$x'(0) = 0$$
$$y(0) = 0$$
$$y'(0) = -2.001584106$$

By converting the two second order differential equations into four first order equations, we can use `solve_ivp` to solve the system over the range $0 \leq t \leq 17$ which is almost the full period for the orbit.



One of the homework exercises asks you to try to convert the two second order ODE's of this Arenstorf system into a set of four first order ODE's, so that they can be solved by `solve_ivp()`.