Systems of Differential Equations
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_systems/ode_systems.pdf

---

A system of differential equations models predation.

"Systems of ODEs"

- *An ordinary differential equation:* $\frac{dy}{dt} = f(t,y)$;
- *An initial value problem gives us* $f(t,y)$ *and the value* $y_0 = y(t_0)$;
- *An ODE describes instantaneous change, dt is infinitesmal;*
- *A computational solver will take discrete time steps* Δt.
- *Euler starts at* y_0 *at time* t_0, *and estimates* y_1 *at* t_1.
- *The error in this approximation depends on the stepsize* Δt.
- *We can try to control this error by comparing two estimates.*

1 Python `lambda` functions

In the previous class, we created the function `euler_solve()`, which could approximately solve a differential equation using Euler's method.

```
def euler_solve ( dydt, tspan, y0, n ):
....
return t, y
```

We gave the name `dydt` to the first input to this function, and in our examples, this was the name of a function file that evaluated the right hand side of the differential equation.

If this right hand side is simple, Python offers an alternative way to specify `dydt`, using what is called a *lambda function*. We have seen various right hand sides, including $f(t,y) = y * (1 - y)$, $f(t,y) = 2t$,

$f(t,y) = 0.2 * t + 0.125 * y$ and $f(t,y) = np.cos(t)$. Instead of referring to a function file, we can define the right hand sides as a formula:

```
t, y = euler_solve ( lambda t, y: y * ( 1 - y ),        [0.0, 5.0], 0.2, 20 )
t, y = euler_solve ( lambda t, y: 2 * t,                [0.0, 2.0], 0.0, 10 )
t, y = euler_solve ( lambda t, y: 0.2 * t + 0.125 * y, [-10.0, 10.0], 5.0, 40 )
t, y = euler_solve ( lambda t, y: np.cos ( t ),         [0,2*np.pi], 0.0, 41] )
```

The format of these descriptions of the right hand side begins with the symbol `lambda` followed by the variables `t` and `y`, followed by a colon, followed by the formula. You can live without ever using this notation, but if you see Python programs written by other people, you will run across this for sure.

A lambda function really is a function. You can even evaluate it like a function if you surround the definition with parentheses, and follow it with the argument value, also in parentheses. For example,

```
(lambda t: t**2 + 1 ) ( 5 )
```

will return the value 26.

2 When More Than One Thing Changes

We have considered differential equations which describe the change over time t of some quantity $y(t)$ by giving its value y_0 at some initial time t_0, and then giving a formula $\frac{dy}{dt} = y'(t) = f(t,y)$, which mathematically should allow us to know the value of $y(t)$ at all future times.

A more complicated system might involve several variables, so that we think of y at any one time as a vector of size say d. In this new situation, we will need, for each component of y, an initial condition and a differential equation describing its rate of change. Thus the initial condition is a vector, the derivative $\frac{dy}{dt}$ is a vector, and each component of the function $f(t,y)$ may now depend on any and all components of the solution.

We can still apply the Euler method to this vector problem, as long as we make the appropriate adjustments. These adjustments are best explained by presenting a modified version of our code, now called *euler_system()*:

```
def euler_system ( dydt, tspan, y0, n ):

  import numpy as np

  m = len ( y0 )                    # Need m so we can properly allocate y

  t = np.linspace ( tspan[0], tspan[1], n + 1 )
  dt = t[1] - t[0]
  y = np.zeros ( [ n + 1, m ] )     #  y is now a 2D array

  for i in range ( 0, n + 1 ):
    if ( i == 0 ):
      y[0,:] = y0.copy ( )          #  Have to ".copy()" the values in y0
    else:
      y[i,:] = y[i-1,:] + dt * dydt ( t[i-1], y[i-1,:] )
                                    #  References to y need 2 indices now
  return t, y
```

So the logic for our revised Euler code is the same. We just need to help Python to understand that the information in the `y` array is now two dimensional.

We will start by considering a system with just 2 variables, called the *predator-prey* or *Lotka-Volterra* equations.

3 The predator-prey system

The predator-prey equations are a simple ecological model in which two species interact. The prey species (maybe rabbits) tends to reproduce at a certain rate; however the population can also change negatively, because each rabbit has a chance of encountering a predator, and getting eaten. Meanwhile, the predator species (perhaps foxes) tendency to die if unfed, but to prosper whenever it encounters a prey to eat.

We might use the symbols `r` for rabbits and `f` for foxes, and then write a pair of coupled differential equations describing the evoluation of the populations:

$$\frac{dr}{dt} = \alpha \cdot r - \beta \cdot r \cdot f$$
$$\frac{df}{dt} = -\gamma \cdot f + \delta \cdot r \cdot f$$

where the parameters (all positive) are:

- α measures the reproductive rate of the rabbits.
- β measures the "cost" to a rabbit of a fox-rabbit encounter.
- γ measures the starvation rate of the foxes.
- δ: measures the "benefit" to the fox of a fox-rabbit cncounter.

We will take these values to be $\alpha = 2.0, \beta = 0.001, \gamma = 10, \delta = 0.002$. We will take the initial conditions $t_0 = 0, r_0 = 5000, f_0 = 100$. Finally, we will be interested in studying this system over the time interval $0 \leq t \leq 5.0$.

Now that we are studying two variables, we need to pack them into a single array y. Similarly, our derivative routine must return an array of two derivatives. Here is how we do that:

```
def predator_prey_dydt ( t, y ):
  import numpy as np

  global alpha, beta, gamma, delta

  r = y[0]
  f = y[1]

  drdt =   alpha * r - beta  * r * f
  dfdt = - gamma * f + delta * r * f

  dydt = np.array ( [ drdt, dfdt ] )

  return dydt
```

Listing 1: Right hand side for predator-prey ODE

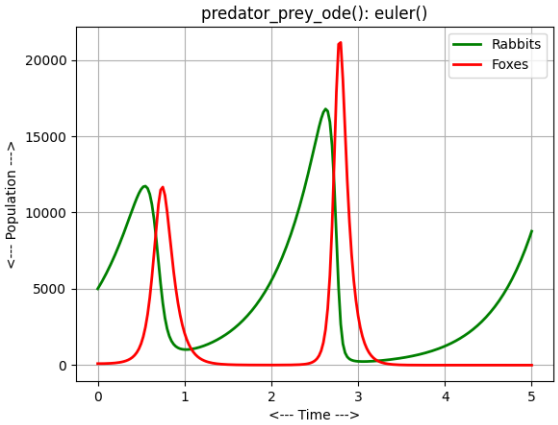
Similarly, when we call `euler_system()`, we must set the initial condition as an array. `y0 = np.array ( [ 5000, 100 ] ).`

We really don't know how to choose a good value of `n`, so we have to experiment.

```
def predator_prey_euler ( ):

  from euler_system import euler_system
  import matplotlib.pyplot as plt
  import numpy as np

  global alpha, beta, gamma, delta

  alpha = 2.0
```

```
beta = 0.001
gamma = 10.0
delta = 0.002

t, y = euler_system ( predator_prey_dydt, [ 0.0, 5.0 ], [ 5000.0, 100.0 ], 200 )
```

Watching how the solution estimates evolve as we increase n actually gives us a decent idea of when things are settling down. By repeatedly increasing this value, at around `n=200`, we get a solution plot that suggests an interesting pattern in the data:



Euler approximation to predator-prey equations, 200 steps.

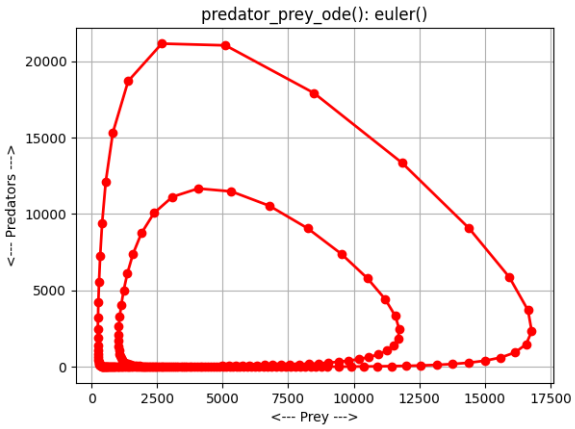
It looks as though both populations are oscillating, perhaps with a regular period, but with growing amplitude. Does this suggest that the population sizes will spiral out of control? In order to believe our results, and to try to improve the accuracy, we need to repeat the calculation, perhaps with 400 steps, and see if the pattern persists.

4 Phase plots

When you see a pair of variables that seem to oscillate together, you might think of them as a version of the sine and cosine functions. Indeed, especially in many physics problems, this behavior is very common. It turns out that our predator prey ODE solution can be regarded as a similar sort of oscillation.

Our time plots paired $t, r(t)$ and $t, f(t)$, showing rabbits and foxes over time. Another way to think about the situation is to show rabbits versus foxes directly, that is, to plot the pairs of values $(r(t), f(t)$ in what is called a *phase plane plot*. For a sine/cosine pair, we'd expect to see a perfect circle. For any sort of periodic behavior, we'd look for a closed curve. For our problem, where we don't think we've gotten an accurate solution yet, we are just interested to see whether the pair of variables is approaching a closed curve.

Here is the result of our phase plot for the 200 step solution:



Euler approximation to predator-prey equations, 200 steps.

Now this looks very suggestive and interesting. Our main question is, if we try for a more accurate solution, will the two loops merge, suggesting that the behavior really is periodic? Or if we go further in time, will bigger and bigger loops appear, getting further apart?

5 Conservation Laws

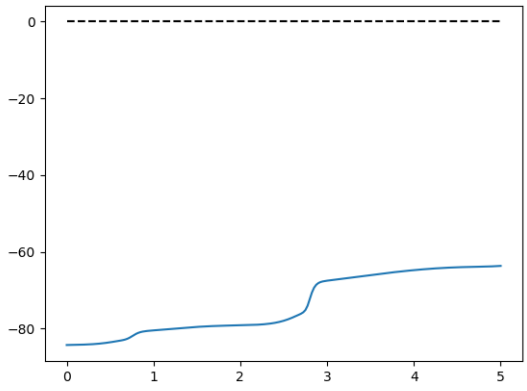
Some systems that are describe by differential equations also satisfy some kind of conservation law. Atoms don't disappear, total energy is not gained or lost, the refrigerant fluid running in a closed system of pipes moves, but doesn't vanish. We are often concerned about the accuracy of ODE solutions we are estimating. If valid solutions satisfy a conservation law, then it's a good idea to see if our estimates do so as well.

Surprisingly enough, it can be shown mathematically that any correct solution $(r(t), f(t)$ of the predator prey system we are considering must also satisfy a conservation law. Suppose we have such a problem with initial conditions $y_0 = y(t_0) = (r(t_0), f(t_0))$ using the parameters $\alpha, \beta, \gamma, \delta$. Define the function

$$h(t) = \delta\, r(t) - \gamma\, \log(r(t)) + \beta\, f(t) - \alpha\, \log(f(t))$$

Then for any time $t_0 <= t$, it must be the case that $h(t) = h(t_0)$, that is, the value of $h(t)$ is constant for all time for the given exact solution.

So, as a check on our solution, we can compute and plot the values of $h(t)$. If this quantity seems to be staying roughly constant, we can assume we are doing a reasonable job of approximating the solution.



Predator-prey euler solution, conservation results, 200 steps.

The results show that our solution is not conserving properly. This gives us another reason to suspect that we are not computing a solution that is accurate enough.

6 Our solution is not accurate enough!

We have several pieces of evidence that our solution is not accurate enough, given that we see our solution "trying" to be more regular.

- The time plots are trying to be periodic, but keep growing in magnitude;
- The phase plot tries to create a closed curve, but instead makes a bigger one;
- The conservation plot shows that our solution is noticeably not conservative;

This suggests that it would be worth our while to try to get a more accurate solution. We can try to achieve that by

- Simply using more steps in the Euler solver;
- Writing a more accurate ODE solver, such as Runge-Kutta 2 or the midpoint method;
- Using a system ODE solver, such as `solve_ivp()`;

7 One Second Order ODE becomes two First Order ODE's!

Especially in physics, we encounter many systems which are described in terms of second derivatives. Newton's laws involve acceleration, which is the second derivative of position, for instance. Electrical systems can involve second derivatives as well. A famous example is known as *van der Pol's equation*

$$\begin{aligned} u'' - \mu(1-u^2)u' + u &= 0 \\ u(0) &= 1 \\ u'(0) &= 0 \end{aligned}$$

where the parameter μ is a damping factor. If $\mu = 0$, then we have a simple harmonic oscillator. We will try to solve this equation over the interval $0 \leq t \leq 10$, for the successive values $\mu = 0.0, 1.0, 2.0$.

Our `euler_solve()` function cannot handle this problem, because it involves second derivatives. However, we can introduce a new variable $v(t) = u'(t)$, and rewrite our single second order problem as a pair of first

order equations, by replacing u'' by v':

$$\begin{aligned} u' &= v \\ v' &= \mu(1-u^2)v - u \\ u(0) &= 1 \\ v(0) &= 0 \end{aligned}$$

Now we can set up the right hand side:

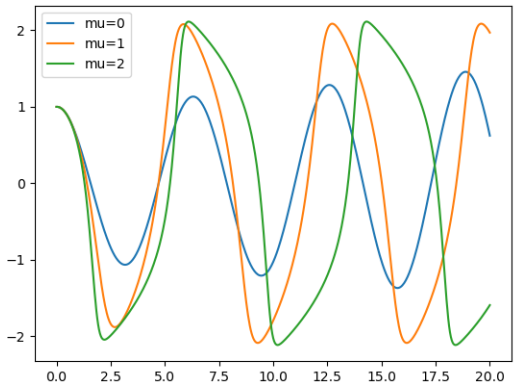
```
def vanderpol_dydt ( t, y ):
  import numpy as np

  global mu

  u = y[0]
  v = y[1]

  dudt =   v
  dvdt = mu * ( 1.0 - u**2 ) * v - u

  dydt = np.array ( [ dudt, dvdt ] )

  return dydt
```

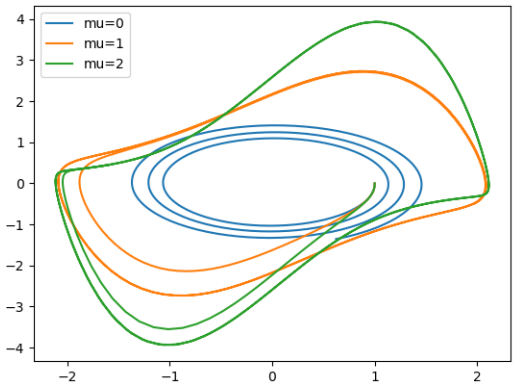
Listing 2: Right hand side for van der Pol ODE

and our main code might look like

```
import matplotlib.pyplot as plt
import numpy as np

global mu

for mu in [ 0.0, 1.0, 2.0 ]:

  t, y = euler_system ( vanderpol_dydt, [ 0.0, 5.0 ], [ 1.0, 0.0 ], 500 )

  plt.clf ( )
  plt.plot ( t, y )
  plt.close ( )

  plt.clf ( )
  plt.plot ( y[0], y[1] )
  plt.close ( )
```

For the time plot, we only display $u(t)$, but not $u'(t)$, for all three values of μ. For the phase plot, we see the circle corresponding to simple harmonic motion when $\mu = 0$, but as μ increases, the phase plot seems to want to become a closed curve, so the solution is periodic, but the shape becomes highly distorted.


mu=0
mu=1
mu=2
2
1
0
−1
−2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


mu=0
mu=1
mu=2
4
3
2
1
0
−1
−2
−3
−4
−2
−1
0
1
2