Euler's ODE Solver
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/euler/euler.pdf

---

The Euler method follows a path by using a sequence of discrete steps.

"Euler's ODE Solver"

- *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 The Initial Value Problem

Physics, biology, and engineering often consider a system which changes over time. In the simplest case, the system at any time t can be characterized by a single measurement $y(t)$. If the value of y changes over time, we expect it does so in accordance with some physical principal. The instantaneous change in the system can be written in terms of a function $f(t, y)$. If we suppose that, at some time t_0, we know the value $y_0 = y(t_0)$, then the *initial value problem* or IVP, is to determine, for $t_0 <= t$, a formula for $y(t)$ which satisfies the pair of conditions

$$y(t_0) = y_0$$
$$\frac{dy}{dy} = f(t, y)$$

If such a formula exists, we call it the analytical solution of the initial value problem.

As an example, consider the logistic IVP. This differential equation can be used to describe a biological population living in an environment with limited resources. There is a natural, maximum population level, and if the population is above or below this level, it tends over time to be driven towards the limiting population size. For a simple case where the maximum population size is 1 (in whatever units we are using), the logistic differential equation can be written as:

$$\frac{dy}{dy} = f(t, y) = y(1 - y)$$

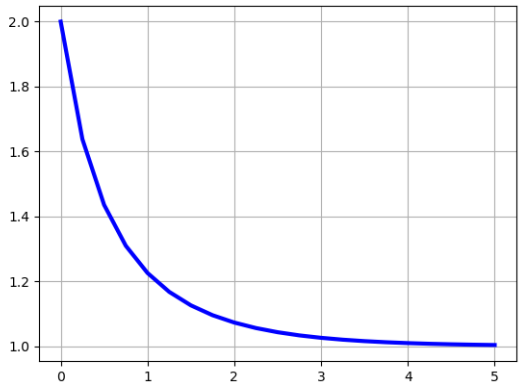
Consider the case where the population at time $t_0 = 0$ is equal to 2. We write this initial condition as

$$y(t_0) = 2$$

and now, it can be shown that the logistic differential equation plus initial condition has the following analytical solution:

$$y = \frac{2e^t}{2e^t - 1}$$

If we plot this solution, we see that the initial population rapidly drops towards the ideal limiting population size of 1.



A solution of the logistic IVP.

Starting from a population size of 0.25 instead, for instance, would result in a plot which rapidly rises towards the value of 1. And actually starting with a population of 1 results in a constant solution.

Determining the solution of a given IVP is a matter of chance, and is generally only successful for textbook example problems. For the IVP's encountered in real life applications, there is generally no analytical solution. If we need the solution to such an ODE, we may have to be willing to accept an approximation.

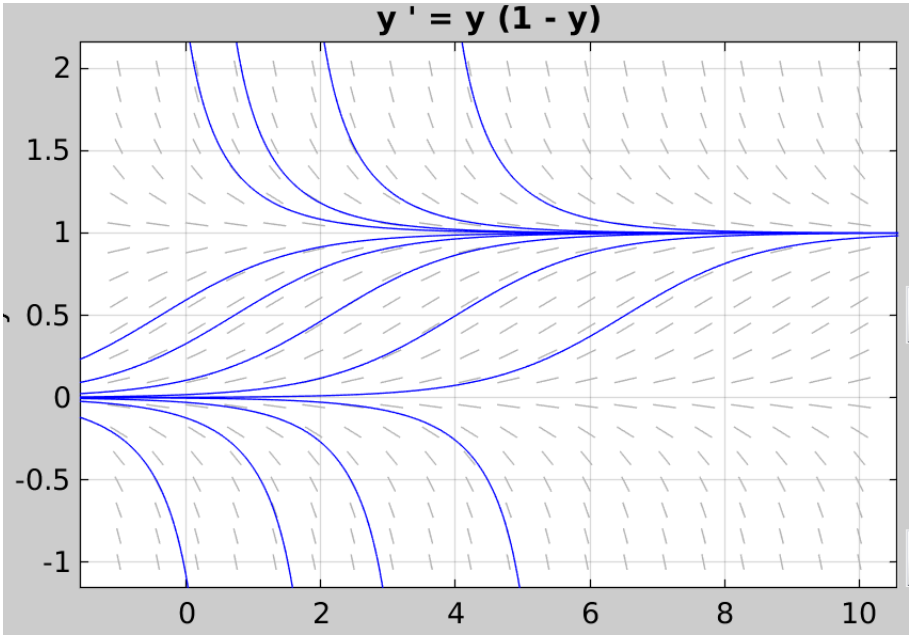
Euler's method was the first tool that could be applied to initial value problems, offering a systematic way of producing estimated values of the solution at discrete times.

2 The $\frac{dy}{dt}$ flow field

To get an understanding of how the solutions to an ODE evolve, one technique is to draw the flow field. To do so, choose a regular grid of (t, y) values, and assign to each grid point the vector $[1, \frac{dy}{dt}]$ or some multiple,

such as $[dt, dy]$. This illustrates a field of vectors that might suggest a weather map of wind direction and strength. In fact, the vectors are like guiding arrows that "tell" an ODE solution to evolve a certain amount in a certain direction.

Consider the following flow field for the logistic ODE, which includes not just the flow vectors, but also some blue lines that represent particular solutions of the IVP. It is clear that if a solution takes on a value between 0 and 1, it will gradually rise to the value 1. Similarly, any solution that takes on a value greater than 1 must rapidly drop down towards 1. Finally, solutions below 0 drop sharply towards $-\infty$. You might also surmise that, although the solutions seem to merge in this plot, all solution lines stay strictly separate.



The flow field for the logistic equation.

We may understand flow fields better if we think about what we would see for some simple cases:

1. $y'(t) = 0$, constant
2. $y'(t) = 1$, linear
3. $y'(t) = t$, quadratic
4. $y'(t) = \cos(t)$, periodic
5. $y'(t) = y$, solutions split apart;

We should plot these examples and discuss their information.

3 Plotting the flow field with Python

It is possible to plot a vector field in Python, using the function `plt.quiver ( T, Y, DT, DY )` where the vector (DT, DY) is to be drawn at the point (T, Y)

Here is an example for our logistic ODE:

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

```
tmin = -1.0
tmax = 10.0
ymin = -0.5
ymax =  1.5

m = 15
n = 21

y = np.linspace ( ymin, ymax, m )
t = np.linspace ( tmin, tmax, n )
dt = t[1] - t[0]
T, Y = np.meshgrid ( t, y )

DT = dt * np.ones ( [ m, n ] )
DY = dt * Y * ( 1.0 - Y )

plt.clf ( )
plt.quiver ( T, Y, DT, DY )
plt.show ( )
```

In this plot, the vectors may have varying lengths. In some cases, it is more helpful to see only the direction of the vectors, and to suppress the varying lengths. To plot the direction field, it is only necessary to normalize the vectors before plotting, as follows:

```
DT = dt * np.ones ( [ m, n ] )
DY = dt * Y * ( 1.0 - Y )
s = np.sqrt ( DT**2 + DY**2 )
DT = DT / s
DY = DY / s
```

Using these ideas, we can easily plot the flow field or direction field for the logistic equation.

Let's modify the plotting code so it can display the fields for the sinusoidal equation $y' = 2\pi \cos(t)$ over the range $0 \leq t \leq 2$.

4 Go with the flow

Our strategy for approximately solving an initial value problem is to start at the given time t_0, and to plan n equally spaced steps of size Δt to t_1, t_2, ..., t_n. We want to fill in a corresponding list of y values, starting from the known quantity y_0, and proceeding to y_n, so that $y_i \approx y(t_0 + i * \Delta t)$.

To estimate y_1, our reasoning is that we know the value of $y(t)$ at t_0, we can evaluate $y'(t_0) = \frac{dy}{dt} = f(t_0, y_0)$ which guarantees that, at least for small steps Δt, we have:

$$y(t_0 + \Delta t) \approx y(t_0) + \Delta t * \frac{dy}{dt}(t_0, y_0)$$

Assuming that the step Δt is small enough, we can then make the estimates

$$\begin{aligned} y_1 &= y_0 + \Delta t\, f(t_0, y_0) \\ y_2 &= y_1 + \Delta t\, f(t_1, y_1) \\ &\ldots \\ y_n &= y_{n-1} + \Delta t\, f(t_{n-1}, y_{n-1}) \end{aligned}$$

and we have filled in our list of estimated solution values.

If we choose a simple ODE, we can easily fill in our estimated solution and compare it to the exact one. Let's take $y'(t) = 2t$, with $y_0 = 0$, whose exact solution is $y(t) = t^2$ and take rather large steps of size 1:

t	y exact	euler formula	euler result	error
0	0	0	0	0
1	1	0 + 1 * (2 * 0)	0	1
2	4	0 + 1 * (2 * 1)	2	2
3	9	2 + 1 * (2 * 2)	6	3
4	16	6 + 1 * (2 * 3)	12	4
5	25	12 + 1 * (2 * 4)	20	5

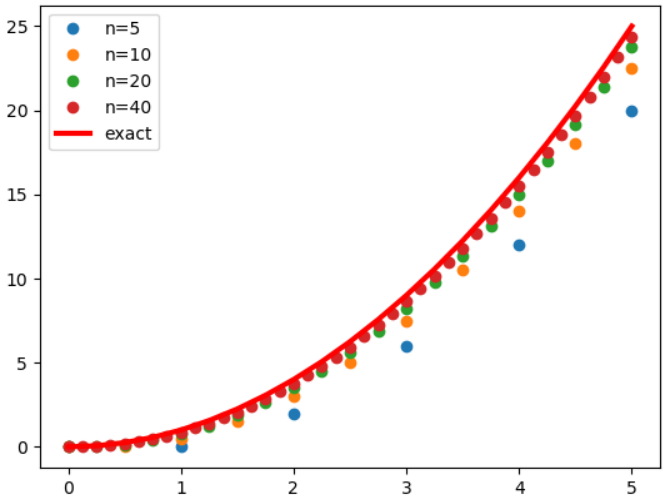
Here we can see that our error is increasing with every step. However, we can get better results by taking smaller steps.

5 A Python Euler Solver

It's easy to use Python to solve our simple quadratic IVP, and we might as well allow ourselves to specify the number of steps to take, so that we can see how the approximation improves.

```
def quadratic_euler_solve ( n )

  tmin = 0
  tmax = 5
  t = np.linspace ( tmin, tmax, n + 1 )
  dt = t[1] - t[0]
  y = np.zeros ( n + 1 )

  for i in range ( 0, n + 1 ):
    if ( i == 0 ):
      y[0] = 0.0
    else:
      y[i] = y[i-1] + dt * ( 2.0 * t[i-1] )

  return t, y
```

We can compare the exact solution of the quadratic IVP to Euler approximations using 5, 10, 20 and 40 steps:



6 Designing an Euler function

Supposing we have an initial value problem, it would be very handy to be able to turn it over to a Python function that will churn out a solution, perhaps using the Euler method. We know that the output will be a set of t, y values. What would the input be? Assume for now that we are satisfied with using a number n of steps of equal size. And we certainly then have to specify the time interval $tmin <= t <= tmax$. And of course, we must give the initial condition $y0$. One more thing is lacking, though. We need to define the right hand side function, which we have symbolized as $f(t, y)$.

If we can figure out how to transmit the right hand side as an input argument, then we could imagine that our Euler ODE solver has for form:

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

where `tspan` is the array `[tmin,tmax]`.

So how do we specify the right hand side `dydt`? For our logistic equation, for instance, we had $f(t, y) = y(1 - y)$; For the quadratic equation, $f(t, y) = 2t$. This information is not a number or a vector; it is a function. How do we describe this to our solver function?

One way is simply to package the right hand side as a Python function, and pass that name to `euler_solve()`. For instance, we could define a right hand side for the logistic equation:

```
def logistic_dydt ( t, y ):
  dydt = y * ( 1 - y )
return dydt
```

in which case, we could invoke our solver by

```
t, y = euler_solve ( logistic_dydt, [0,10], 0.5, 21 ):
```

In other words, we simply pass the name of the right hand side function.

Alternatively, as long as our right hand side is not too complicated, we can describe it using a compressed format known as a `lambda function`. For instance, we could request an Euler solution for our quadratic problem by

```
t, y = euler_solve ( lambda t, y: 2 * t, [0.0, 5.0], 0.0, 41 ):
```

Similarly, for the "tpy" ODE, over the interval $0 \leq t \leq 10$, with initial condition $y_0 = 5$, whose right hand side is

$$y'(t) = \frac{1}{5}t + \frac{1}{8}y$$

we would have

```
t, y = euler_solve ( lambda t, y: 0.20*t+0.125*y, [0.0, 10.0], 5.0, 21 ):
```

The format of the description 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.

7 An Euler ODE Solver

Now that we have settled the input and output of the Euler solver, we can turn to the "throughput" or internal details, of our function. Here, we can start by simply grabbing the guts of our Euler solver for the quadratic ODE and modifying them.

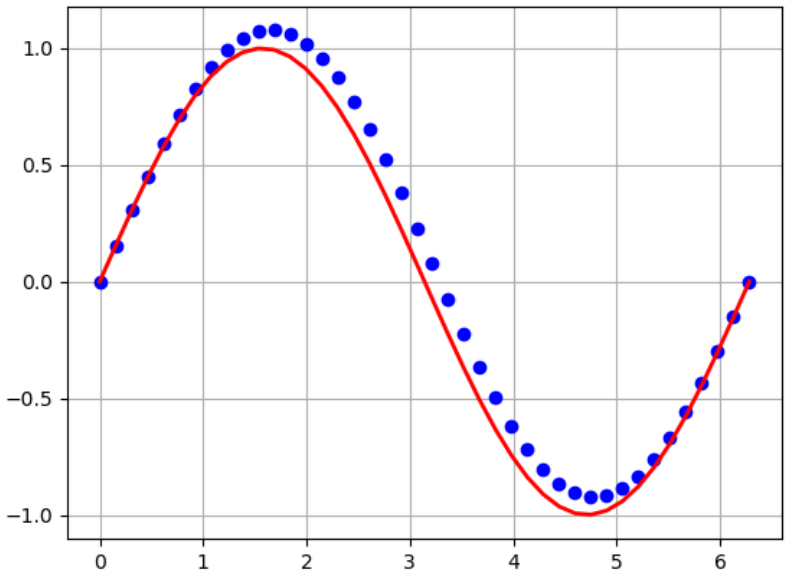
```
def euler_solve ( dydt, tspan, y0, n ):

  t = np.linspace ( tspan[0], tspan[1], n + 1 )
  dt = t[1] - t[0]
  y = np.zeros ( n + 1 )

  for i in range ( 0, n + 1 ):
    if ( i == 0 ):
      y[0] = y0
    else:
      y[i] = y[i-1] + dt * dydt ( t[i-1], y[i-1] )

  return t, y
```

8 Testing the solver

We can easily test `euler_solve()` on some of our examples:

```
t, y = euler_solve ( lambda t, y: np.cos ( t ), [0,2*np.pi], 0.0, 41]
y2 = np.sin ( t )
plt.plot ( t, y, 'bo' )
plt.plot ( t, y2, 'r-' )
```

Here is a plot comparing our Euler estimate and exact solution:



Of course, instead of using a lambda function, we could have written a function file instead:

```
def sinusoidal_dydt ( t, y ):
  import numpy as np
  dydx = np.cos ( t )
  return dydt
```

in which case our call would be:

```
t, y = euler_solve ( sinusoidal_dydt, [0,2*np.pi], 0.0, 41]
```