Differential Equations
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/differential_equations/differential_equations.pdf

A system of differential equations models predation.

Differential Equations

- *Differential equations describe change from an initial state over infinitesimal steps;*
- *Numerical methods approximate this evolution using a sequence of finite steps;*
- *Time and phase plots are important graphical tools to analyze numerical solutions.*
- *Numerical methods reduce higher-order differential equations to a first order system;*
- *A differential equation may include a hidden conservation law.*
- *Some numerical methods can preserve such conserved quantities.*
- *The accuracy of a numerical solution generally decreases over time ;*
- *Using smaller steps, adaptivity, or a higher order method, can improve accuracy;*
- *The* `scipy()` *function* `solve_ivp()` *is a powerful differential equation solver.*

1 A differential equation

An ordinary differential equation (abbreviated as "ODE") describes the change, over time, of some quantity $y(t)$, and is usually written as

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

Usually, there is an additional item of information, known as an initial condition, specifying the value of $y(t)$ at some starting time:

$$y(t_0) = y_0$$

Together, this information poses an *initial value problem.*

In textbooks, there are techniques for solving certain initial value problems, producing a formula for $y(t)$. However, these methods apply to a limited set of problems, and for the remainder we must try numerical methods.

A numerical method will generally produce an approximate solution of the problem, giving a sequence of values (t_i, y_i) that predict the behavior of the solution over some time interval $[t_0, t_{stop}]$.

We will look at a variety of simple initial value problems, show how to write a Python code that can make a rough estimate of the solution, make plots that can illustrate feeatures of the solution, and present a Python function that does an excellent job of producing accurate solutions automatically.

2 The logistic equation

As long as there is plenty of nutrients and other resources, a colony of living organisms will increase in population. It is natural to assume that each member of the colony has a natural rate r of reproduction, so that a colony of size y_0 will, over some period of time, increase by $r * y$ members. This is an example of exponential growth, described by the differentical equation

$$\frac{dy}{dt} = r \cdot y$$

with the exact solution $y(t) = y_0 \cdot e^{rt}$.

But this explosive exponential growth rapidly reaches a barrier, the natural carrying capacity of the environment. As the colony grows, each member has a smaller share of the fixed resources, and this acts to oppose the population increase. Thus, it might be expected that the population will eventually level off at some size where it can just be sustained.

A differential equation which describes this more elaborate population model is known as the *logistic equation.* Along with the growth rate r, we now include a carrying capacity or maximum population size k. Our diffential equation is then

$$\frac{dy}{dt} = r \cdot y \cdot (1 - \frac{y}{k})$$

You can see that, if $y << k$, the behavior is essentially exponential, but as y approaches the limiting value k, the factor $(1 - \frac{y}{k})$ is going to fiercely reduce the growth rate.

If we know the values of r and k, and we have the population size y_0 at time $t_0 = 0$, then there is a formula for the exact solution of this equation:

$$y(t) = \frac{k \cdot y_0 \cdot e^{rt}}{k + y_0 \cdot (e^{rt} - 1)}$$

3 The Euler method

The forward Euler method for solving an ODE is very simple:

- Assume you have a solution estimate `(t,y)`;
- Assume you have chosen a stepsize `dt`;
- Evaluate the ODE right hand side `dy/dt` at `(t,y)`;
- Update your position to `t+dt, y+dy/dt*dt`;
- Repeat as often as desired.

It should be clear that the sequence of solution estimates you get is going to depend a lot on the stepsize `dt` that you choose. For instance, let's consider the exponential growth problem $fracdydt = y$, starting at $t_0 = 0$, $y_0 = 1$. We will use three different stepsizes, and compare the results with the exact solution.

```
           dt       dt      dt
   t      0.25    0.125   0.0625  Exact

 0.000   1.000    1.000   1.000   1.000
```

```
0.062                     1.062   1.064
0.125             1.125   1.129   1.133
0.188                     1.199   1.206
0.250   1.250     1.266   1.274   1.284
0.312                     1.354   1.367
0.375             1.424   1.439   1.455
0.438                     1.529   1.549
0.500   1.562     1.602   1.624   1.649
0.562                     1.726   1.755
0.625             1.802   1.834   1.868
0.688                     1.948   1.989
0.750   1.953     2.027   2.070   2.117
0.812                     2.199   2.254
0.875             2.281   2.337   2.399
0.938                     2.483   2.554
1.000   2.441     2.566   2.638   2.718
```

You can observe that the Euler estimate using `dt=0.25` quickly deviates from the exact solution, but that, as we decrease the stepsize to 0.125 and then 0.0625, the estimates improve. On the other hand, even then, we see that, for every stepsize we tried, the accuracy decreases as we proceed forward in time.

These two observations, that smaller stepsizes improve accuracy, and that the error tends to increase with time, are actually a persistent feature of the numerical solution of differential equations.

4 An Euler code

Since we would like to use the Euler method repeatedly, for a variety of stepsizes, and even for a variety of ODE's, we will want to create a Python function, named *euler.py*, which encapsulates the method.

The function will have the rough form:

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

  import numpy as np

  m = np.size ( y0 )

  t0 = tspan[0]
  tstop = tspan[1]
  dt = ( tstop - t0 ) / n
  t = np.zeros ( n + 1 )
  y = np.zeros ( [ n + 1, m ] )

  for i in range ( 0, n + 1 ):
    if ( i == 0 ):
      t[i]   = t0
      y[i,:] = y0
    else:
      t[i]   = t[i-1]   + dt
      y[i,:] = y[i-1,:] + dt * ( dydt ( t[i-1], y[i-1,:] ) )

  return t, y
```

Listing 1: Outline of an Euler ODE solver

Here, the input variables are:

- `dydt` is the name of a derivative function.

- `tspan` is a vector of length 2. `tspan[0]` is the start, and `tspan[1]` the final time.
- `y0` contains the value of y at the initial time.
- `n` is the number of equal steps to take.

and the output is:

- `t` a vector of length `n+1`, containing the initial time, and the `n` later times.
- `y` is a vector of length `n+1`, containing the initial y0, and the `n` later estimates.

The first part of the code does some initialization, unpacking `tspan`, computing `dt`, setting aside memory for the `t` and `y` vectors. The following loop first stores the initial condition, and then each further step extends the solution vector to a later time.

5 Setting up and solving the logistic ODE

Now let us suppose we are interested in solving the logistic ODE, for which we have parameters $t_0 = 0, y_0 = 0.1, r = 1, k = 1$.

First, we need a function that defines the right hand side:

```python
def logistic_deriv ( t, y ):

  r = 1
  k = 1

  dydt = r * y * ( 1.0 - y / k )

  return dydt
```

Listing 2: The logistic right hand side function.

Since we know the exact solution, it will be useful to have another function available to evaluate that:

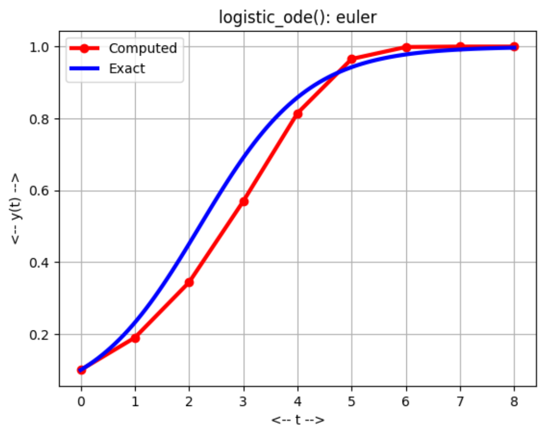
```python
def logistic_exact ( t ):

  r = 1
  k = 1
  y0 = 0.1

  y = ( k * y0 * np.exp ( r * t ) ) \
    / ( k + y0 * ( np.exp ( r * t ) - 1.0 ) )

  return y
```

Listing 3: The logistic exact solution function.

Now we are ready to call the Euler solver:

```python
t, y = euler ( logistic_deriv , [0.0,8.0], 0.1, 8 )
plt.plot ( t, y )
t2 = np.linspace ( 0.0, 8.0, 101 )
y2 = logistic_exact ( t2 )
plt.plot ( t, y, 'ro' )
plt.plot ( t2, y2, 'b-' )
plt.show ( )
```

Listing 4: Euler solution of logistic equation.



Solving the logistic equation with Euler's method.

The plot suggests that our code has the right idea about the logistic equation, but has made some noticeable errors. We can assume that these errors would decrease if we used more steps.

6 The stiff equation

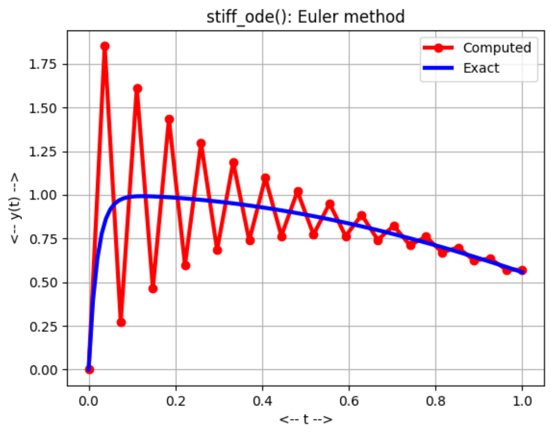
Consider the "stiff" differenatial equation defined by:

$$\frac{dy}{dt} = 50 \cdot (\cos(t) - y)$$
$$y(0) = 0$$

The exact solution is:

$$y(t) = 50 \cdot \frac{\sin(t) + 50 \cdot \cos(t) - 50 \cdot e^{-50 \cdot t}}{50^2 + 1.0}$$

We are interested in solving this problem over the interval $0 \leq t \leq 1.0$. We will use our Euler code, and take $n = 27$ steps. When we plot the approximate and exact solutions, the results are unpleasant:



Solving the stiff equation with Euler's method.

From the plot, we can see a suggestion for where the trouble comes from. The slope of the exact solution at the origin is almost vertical. The Euler step follows this direction, whereas the slope of the exact solution decreases. Thereafter, the approximate solution seems to struggle to get back on track, but continues to zigzag back and forth across the exact solution. Luckily for us, these oscillations gradually die down and the error drops steadily.

Even if we didn't know the exact solution, the behavior of our approximation would suggest that something is wrong. If our stepsize is about right, then repeating the computation with a smaller stepsize should give about the same picture. So looking at the plot of our approximation, we should naturally expect that using more steps would be necessary.

7 The predator-prey equations

The predator-prey equations are a simple ecological model in which two species interact. The prey species tends to reproduce at a certain rate, but also to be eaten whenever it encounters a predator. Meanwhile, the predator species has a 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 $0 \leq t \leq 5.0$.

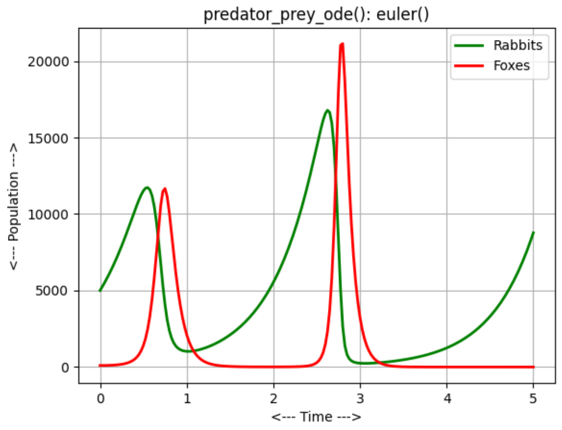
Now that we are studying two variables, we need to pack them into a single array y. Similarly, our derivative routine must return an array of two derivatives. Here is how we do that:

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

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

  alpha = 2.0
  beta = 0.001
  gamma = 10.0
  delta = 0.002

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

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

  return dydt
```

Listing 5: Right hand side for predator-prey ODE

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

We really don't know how to choose a good value of `n`, so we have to experiment. By repeatedly increasing this value, at around `n=200`, we get a solution plot that suggests an interesting pattern in the data:



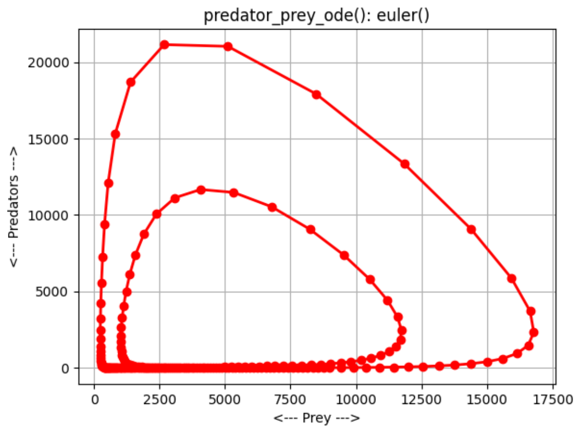
Euler approximation to predator-prey equations, 200 steps.

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.

8 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. While the time plot that we made suggests this, a better way to analyze what is going on is to make a *phase plot*, in which we plot $r(t)$ on the x axis and $f(t)$ on the y axis. 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 what is going on.

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?

9 Using the scipy function solve_ivp()

If this were a course in ODE's, we would continue to explore other solution techniques, including the backward Euler method, the midpoint method, BDF2, and Runge Kutta methods. However, we assume that you are less interested in writing your own ODE solver, and more interested in solving ODE's. The `scipy` library includes a powerful function ttsolve_ivp() which is very good at efficiently and accurately approximating ODE solutions. Let us consider how we would use it to deal with the problems we have see so far.

The simplest calling sequence for the ODE solver has the form

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

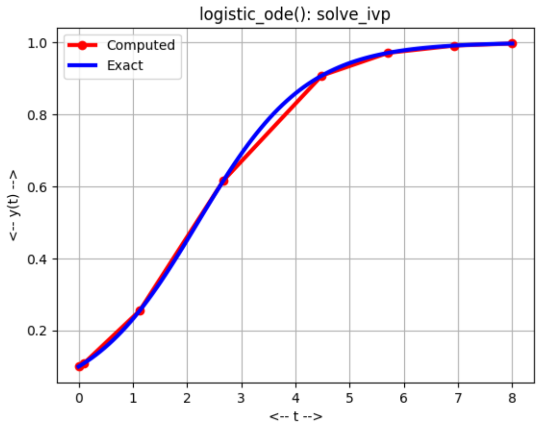
The input argument list has the same meaning as we have used for the `euler()` function. However, we no longer supply `n`, the number of steps. Also, the value `y0` must be an array, even if it only has dimension 1. That's because `solve_ivp()` is an *adaptive* ODE solver, which varies the stepsize, taking small steps at difficult times, and larger steps when it seems safe.

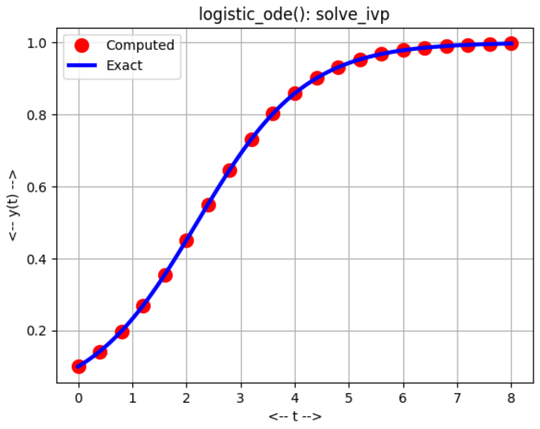
The output `sol` is different from what we have seen before. It is a sort of structure, storing a number of components. In particular, if we are solving a scalar problem (like the logistic equation), then our solution values are accessible as the array `sol.y0[0]`; for the predator-prey problem, we would be looking at the arrays `sol.y0[0]` for the rabbits, and `sol.y0[1]` for the foxes. The actual time values that were computed are stored as `sol.t[]`.

Thus, here is a code to solve the logistic equation using this solver:

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

sol = solve_ivp ( logistic_deriv, [0.0,8.0], [ 0.1 ] )
plt.plot ( sol.t, sol.y[0] )
t2 = np.linspace ( 0.0, 8.0, 101 )
y2 = logistic_exact ( t2 )
plt.plot ( t, y, 'ro' )
plt.plot ( t2, y2, 'b-' )
plt.show ( )
```

Listing 6: Euler solution of logistic equation.





Logistic solutions by `solve_ivp()`.

When we plot the solution from `solve_ivp()` something seems wrong. There are huge gaps between the exact and computed solutions. But in fact, this is just because we made a bad plot. Notice that all the red dots lie right on the exact curve. The straight lines connecting these dots are not part of the solution, they are just meant to suggest how the curve looks. The `solve_ivp()` code just didn't need to take many steps to get an accurate solution.

But if we want to see more detail, we can ask the code to go back and fill in the solution values at intermediate points. We do that by using an additional `t_eval = ?` argument. To get the second plot above, we compute a list of where we want the solutions, and then add that list on our input to `solve_ivp()`. Everything else stays the same:

```
t1 = np.linspace ( 0.0, 8.0, 21 )
sol = solve_ivp ( logistic_deriv , [0.0,8.0], [ 0.1 ], t_eval = t1 )
```

Listing 7: Euler solution of logistic equation.

Now `sol.t` and `sol.y[0]` will be evaluated at 21 equally spaced points in the interval, and our plot will give us a much better idea of how well the computed solution matches the exact solution.

10 The pendulum

A pendulum of length l swings back and forth under the influence of gravity with a force g, making an angle θ with the vertical axis. For small angles θ, the behavior of the pendulum can be approximated by the second order differential equation

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l} \cdot \theta$$

If the pendulum is initially at the angle θ_0 with zero velocity $\theta_0' = 0$, then the exact solution is

$$\theta(t) = \theta_0 \cdot \cos(\sqrt{\frac{g}{l}} \cdot t)$$

and therefore the pendulum swings with a period of

$$p = 2 \cdot \pi \cdot \sqrt{\frac{l}{g}}$$

Now we'd like to use `solve_ivp()` here, but that code does not work with a second order ODE. Thus, we have to rewrite this second order system as a pair of first order equations. Writing

$$\begin{aligned} u &= \theta \\ v &= \theta' \end{aligned}$$

we have, by differentiation,

$$\begin{aligned} u' &= \theta' = v \\ v' &= \theta'' = -\frac{g}{l} \cdot \theta = -\frac{g}{l} \cdot u \end{aligned}$$

or, in short

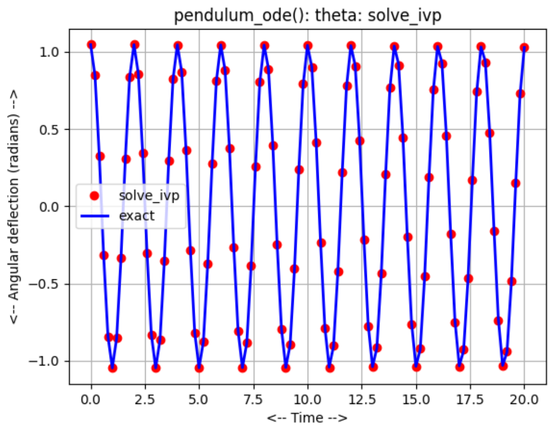
$$\begin{aligned} u' &= v \\ v' &= -\frac{g}{l} \cdot u \end{aligned}$$

and now we can pass `solve_ivp()` the vector `y = [ u, v ]` and similarly write the right hand side code in terms of a two-dimensional variable `y`.

Now we take as parameters $g = 9.81, l = 1, t0 = 0, y_0 = [\frac{\pi}{3}, 0]$ and wish to integrate over $0 \le t \le 20$. Our right hand side code will be

```
def pendulum_deriv ( t, y ):

  import numpy as np

  g = 9.81
  l = 1

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

  dudt = v
  dvdt = - ( g / l ) * u

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

  return dydt
```

Listing 8: right hand side of pendulum ODE.



The pendulum equation. The blue line is the exact solution, the red dots the `solve_ivp()` *solution points.*

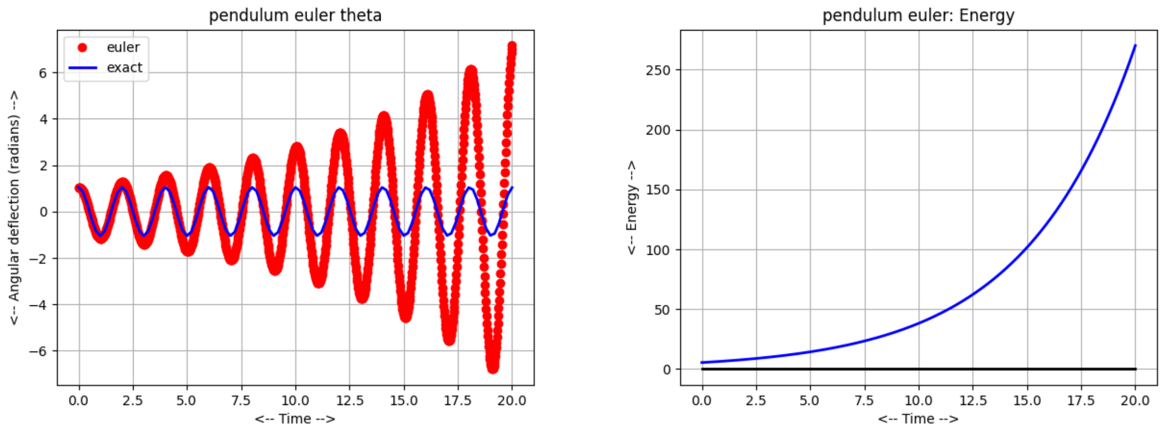
11 Conservation laws

Many ODE's describe the behavior of a physical system for which a conservation law might apply. For the pendulum problem, since there is no forcing term, and we are not modeling the effect of friction, the energy of the pendulum should be constant over time. Again using `u` and `v` for the values of θ and θ', and assuming that the pendulum bob has a mass $m = 1$, the energy at time t should be the sum of potential and kinetic energies:

$$E(t) = \frac{1}{2} \cdot m \cdot g \cdot l \cdot u(t)^2 + \frac{1}{2} \cdot m \cdot v(t)^2$$

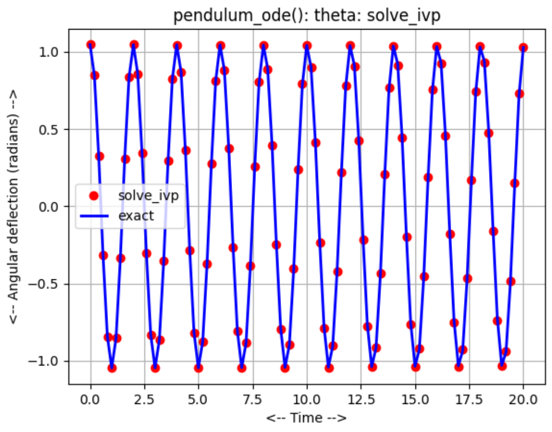
Even if we don't know the exact solution of our problem, we can monitor and plot the value of $E(t)$. If it deviates significantly from its initial value, then we know that our ODE solver is doing a poor job.

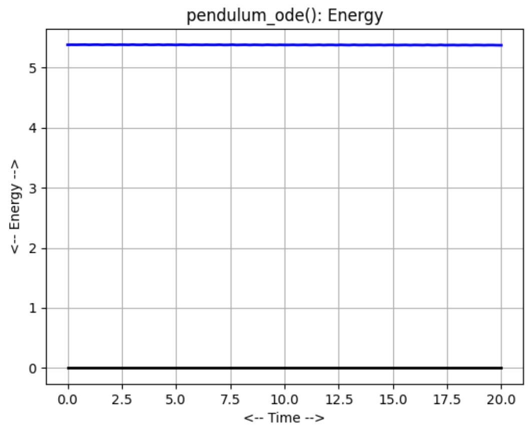
Here, for instance, are the plots of $\theta(t)$ and $E(t)$ for the pendulum problem, solved by the Euler method with 1000 fixed-size steps:



Euler approximation to $\theta(t)$ and $E(t)$ equations, 1000 steps.

and here are the corresponding plots from `solve_ivp()`, for which we requested 101 equally spaced plot points:





`solve_ivp()` *approximation to* $\theta(t)$ *and* $E(t)$ *equations.*

Although the plot of θ returned by the Euler code might seem like a reasonable solution, the explosion in the value of the energy should be an alarm warning that this result is not a good approximation. In cases where an ODE system satisfies a conservation law, it is always worth while to compute and monitor that quantity as an extra measure of accuracy.

12 Exercises

1. Consider a biological system with growth rate $r = 0.25$ and carrying capacity $k = 100$. Suppose that at time $t = 0$ the population is $y_0 = 10$. Plot the population predicted by the exponential and logistic models, over the time period $0 \leq t \leq 10$. Draw the two curves on the same plot, one curve in red and one in blue.
2. In the text, the logistic equation, with parameters $t_0 = 0, y_0 = 0.1, r = 1, k = 1$ was solved from $0 \leq t \leq 8$ using 8 steps. Recompute this solution using 16, and 32 steps. Show all three of these solutions on a single plot, along with the exact solution. You should see evidence that increasing the number of steps produces a better approximation to the exact solution.
3. For the logistic equation, the parameter r determines how steeply the curve rises during the exponential growth phase. Use the parameters $t_0 = 0, y_0 = 0.1, k = 1$, interval $0 \leq t \leq 8$ and 8 steps. Solve the problem 3 times, with parameter r set to 1, 2 and 4. For each solution attempt, compare the Euler solution and the exact solution. What do you imagine will happen if you continue to increase r?
4. For the logistic equation, we expect solutions to remain between 0 and k, the maximum carrying capacity or maximum sustainable population. But let us investigate what happens if we have an initial condition that is *above* this value. Use the parameters $t_0 = 0, y_0 = 2.0, r = 1, k = 1$, interval $0 \leq t \leq 8$ and 8 steps. Compare the Euler solution and the exact solution.
5. For the logistic equation, what will happen if our initial condition y_0 is negative?
6. For the logistic equation, what is special about the initial conditions $y_0 = 0$ and $y_0 = 1$?
7. For the stiff equation, the solution with $n = 27$ Euler steps is clearly a disaster. Experiment with the value of n, and try to produce an approximation whose plot is noticeably closer to the exact solution.
8. For the predator-prey equation, repeat the calculation using 400 steps. Plot the solutions and compare them to the solution for 200 steps. What does this comparison suggest?
9. For the predator-prey equation, repeat the calculation using 400 steps. Make another phase plot, and see whether the two loops come closer together. Then try extending the solution to the interval $0 \leq t \leq 10$, using 800 steps. Your phase plot should now include more loops. Do they stay close together, or do they split apart?

10. For the stiff equation, use solve_ivp(), recompute the solution, and plot it against the exact solution. Our bad Euler solution used 27 steps. Can you figure out how many steps solve_ivp() used, just by looking at the plot?
11. For the predator prey equations, use solve_ivp(), recompute the solution, and plot the phase plane. The successive loops should lie almost exactly on top of each other. However, if solve_ivp() takes large steps, your plot might not seem to be lining up well. In that case, try using the t_eval input argument to request 1001 equally spaced solution values in the time interval.
12. For the pendulum equation, compute the phase portrait of the solution.
13. The nonlinear pendulum equation is $\frac{d^2\theta}{dt^2} = -\frac{g}{l} \cdot \theta$ Use solve_ivp() to solve this equation. Compare the linear and nonlinear solutions on a single plot. Differences will be more obvious if the initial condition θ_0 is large.
14. The predator-prey problem has a conserved quantity:

$$E(t) = \delta \cdot r(t) - \gamma \cdot \log(r(t)) + \beta \cdot f(t) - \alpha \cdot \log(f(t))$$

Use euler() and solve_ivp() to solve the equation, compute and plot $E(t)$ and observe whether $E(t)$ stays essentially constant over time for each solver.