Lab 3: Implicit ODE methods
MATH2071, University of Pittsburgh, Spring 2023

1 Introduction

The explicit methods that we discussed last time are well suited to handling a large class of ODE's. These methods perform poorly, however, for a class of "stiff" problems that occur all too frequently in applications. We will examine *implicit methods* that are suitable for such problems. We will find that the implementation of an implicit method has a complication we didn't see with the explicit method: a (possibly nonlinear) equation needs to be solved.

The term "stiff" as applied to ODE's does not have a precise definition. Loosely, it means that there is a very wide range between the most rapid and least rapid (with x) changes in solution components. A reasonably good rule of thumb is that if Runge-Kutta or Adams-Bashforth or other similar methods require much smaller steps than you would expect, based on your expectations for the solution accuracy, then the system is probably stiff. The term itself arose in the context of solution for the motion of a stiff spring when it is being driven at a modest frequency of oscillation. The natural modes of the spring have relatively high frequencies, and the ODE solver must be accurate enough to resolve these high frequencies, despite the low driving frequency.

This lab will take three sessions.

2 Stiff Systems

I've warned you that there are problems that defeat the explicit Runge-Kutta and Adams-Bashforth methods. In fact, for such problems, the higher order methods perform even more poorly than the low order methods. These problems are called "stiff" ODE's. We will only look at some very simple examples.

Consider the differential system

$$\begin{aligned} y' &= \lambda(-y + \sin x) \\ y(0) &= 0 \end{aligned} \tag{1}$$

whose solution is

$$\begin{aligned} y(x) &= Ce^{-\lambda x} + \frac{\lambda^2 \sin x - \lambda \cos x}{1+\lambda^2} \\ &= Ce^{-\lambda x} + \frac{\lambda^2}{1+\lambda^2}\sin x - \frac{\lambda}{1+\lambda^2}\cos x \end{aligned}$$

for C a constant. For the initial condition $y = 0$ at $x = 0$, the constant C can easily be seen to be

$$C = \frac{\lambda}{1+\lambda^2}$$

The ODE becomes stiff when λ gets large: at least $\lambda = 10$, but in practice the equivalent of λ might be a million or more. One key to understanding stiffness of this system is to make the following observations.

- For large λ and except very near $x = 0$, the solution behaves as if it were approximately $y(x) = \sin x$, which has a derivative of modest size.
- Small deviations from the curve $y(x) = \sin x$ (because of initial conditions or numerical errors) cause the solution to have large derivatives that depend on λ.

In other words, the interesting solution has modest derivative and should be easy to approximate, but nearby solutions have large (depending on λ) derivatives and are hard to approximate. Again, this is characteristic of stiff systems of ODEs.

3 stiff_ode()

As an example of a stiff ODE, we will consider the example problem

$$\frac{dy}{dx} = \lambda(-y + \sin(x))$$

where λ is a parameter whose value controls the stiffness of the ODE.

Create the corresponding file `stiff_ode.py`:

```
def dydx = stiff_ode ( x, y ):
  """
  dydx = stiff_ode ( x, y ) right side of the stiff ODE

  input:
    x is independent variable
    y is dependent variable
  output:
    dydx is the value of f_ode(x,y).
  """
  import numpy as np
  LAMBDA = 4.0
  dydx = LAMBDA * ( - y + np.sin ( x ) )
  return dydx
```

4 stiff_solution()

The solution to our stiff ODE is

$$y = \frac{\lambda^2}{\lambda^2 + 1}\sin(x) + \frac{\lambda}{\lambda^2 + 1}(e^{-\lambda x} - \cos(x))$$

Add the following definition of `stiff_solution()` to your file `stiff_ode.py`:

```
def stiff_solution ( x ):
  """
  y = stiff_solution ( x ), solution of the stiff ODE.

  input:
    x is the independent variable
  output:
    y is the solution value
  """
  import numpy as np
  LAMBDA = 4.0
  y = ( LAMBDA**2 / ( 1 + LAMBDA**2 ) ) * np.sin(x) + \
      ( LAMBDA    / ( 1 + LAMBDA**2 ) ) * ( np.exp ( - LAMBDA * x ) - np.cos ( x ) )
  return y
```

5 Exercise 1

1. Create a file `exercise1.py` for this exercise.
2. Use `forward_euler()` from the previous lab to solve the stiff ODE, using `y(0.0) = 0.0`, and solve up to `x` $= 2\pi$. Try using `numSteps = 31`.
3. Evaluate the exact solution at 101 evenly spaced points in $[0.0, 2\pi]$.
4. Plot the ODE approximation and the exact solution. Presumably, these will be quite similar.

6 Making it easy to vary LAMBDA

It turns out that the value of λ has an important influence on the approximate solution of the stiff ODE. In order to investigate this, it will be convenient to be able to consider several values for the variable `LAMBDA`. Of course one way to do this is simply to modify the lines `LAMBDA=4.0` to the desired value as many times as necessary.

Another way is to create a function which can set, reset, and remember the value of `LAMBDA`. That way, a single program can run once, and choose LAMBDA to be 4, 55, and 1000 in succession. (These are the values we will actually be interested in.) To do this requires writing an *awfully mysterious* function, but one which is pretty easy to use. And we will have to modify the two functions in `stiff_ode.py` to work with this new idea.

To begin with, let's update `stiff_ode.py`. There are two places where the value of `LAMBDA` is set equal to 4. Replace each of those lines by the following

```
LAMBDA = stiff_lambda ( )
```

Then we need to insert the code for `stiff_lambda()` into the `stiff_ode.py` file.

7 stiff_lambda()

Add the following function definition to your file `stiff_ode.py`:

```
def stiff_lambda ( lambda_input = None ):

  if not hasattr ( stiff_lambda, "lambda_default" ):
    stiff_lambda.lambda_default = 4.0

  if ( lambda_input is not None ):
    stiff_lambda.lambda_default = lambda_input

  lambda_output = stiff_lambda.lambda_default

  return lambda_output
```

You don't need to understand how this function does its magic. But it means that you can easily change the value of LAMBDA at any time, and all your codes will be able to find out what that new value is.

If you want to use `stiff_lambda()` outside of the `stiff_ode.py` file, then you have to import it:

```
from stiff_ode import stiff_lambda
```

The function starts out assuming that LAMBDA is 4.

To find out the current value of LAMBDA, you write

```
  LAMBDA = stiff_lambda ( )
```

To change the current value of LAMBDA, you write

```
  stiff_lambda ( LAMBDA )
```

To see whether you've set things up correctly, rerun exercise 1, after you issue the commands:

```
from stiff_ode import stiff_lambda
LAMBDA = 1.0
stiff_lambda ( LAMBDA )
```

Your plot of the exact and approximate solutions should have roughly the same shape as for `LAMBDA = 4.0`, but now the maximum value will be much reduced.

8 The direction field of an ODE

One way of looking at a differential equation is to plot its "direction field." At any point (x, y), we can plot a little arrow $\mathbf{p}$ equal to the slope of the solution at (x, y). This is effectively the direction that the differential equation is "telling us to go," sort of the "wind direction." How can you find $\mathbf{p}$? If the differential equation is $y' = f_{\text{ode}}(x, y)$, and $\mathbf{p}$ represents y', then $\mathbf{p} = (dx, dy)$, or $\mathbf{p} = h(1, f_{\text{ode}}(x, y))$ for a sensibly-chosen value h. There are built-in functions to generate this kind of plot.

The solution to our stiff ODE is roughly $\sin x$. At every point $(x, y(x))$, the solution curve is tangent to the value of the derivative function $f(x, y(x)) = \frac{dy}{dx}$. Basically, solving the ODE is equivalent to correctly connecting the sequence of these tangent directions. How hard this task can be depends on the behavior of the entire direction field, that is, the plot of $f(x, y)$ for all values of (x, y), not just for the $y(x)$ that satisfies our ODE. This direction field not only tells us how to follow the solution curve. It also tells us the shape of solution curves that have nearby initial values. It tells us what will happen if we make a small error, and wind up on a nearby solution curve. And it suggests what will happen to an ODE solver that must approximate the continuous solution by taking relatively large steps according to the direction field.

9 stiff_direction()

Create a file `stiff_direction.py`, and insert the following statements, which will allow you to plot the direction field given an input value of `LAMBDA`:

```
def stiff_direction ( LAMBDA ):

  from stiff_ode import stiff_ode, stiff_solution, stiff_lambda
  import matplotlib.pyplot as plt
  import numpy as np
#
#  Define the value of LAMBDA.
#
  stiff_lambda ( LAMBDA )
#
#  Sample dy/dx over [0,2pi]x[-1,+1]
#
  x = np.linspace ( 0.0, 2.0 * np.pi, 16 )
  y = np.linspace ( -1.5, +1.5, 9 )
  X, Y = np.meshgrid ( x, y )
  PX = np.ones ( X.shape )
  PY = stiff_ode ( X, Y )
  scale = 5.0
  plt.quiver ( X, Y, PX, PY, scale )
#
#  Evaluate and plot the exact solution.
#
  x2 = np.linspace ( 0.0, 2.0 * np.pi, 101 )
  y2 = stiff_solution ( x2 )
  plt.plot ( x2, y2, 'r' )

  plt.grid ( True )
  plt.axis ( 'equal' )
  plt.title ( 'Direction field for stiff ODE, LAMBDA = ' + str ( LAMBDA ) )
  plt.savefig ( "stiff_direction.jpg" )
  plt.show ( )
  plt.close ( )
```

10 Exercise 2

1. Create a file `exercise2.py` for this experiment.

2. Consider the sequence of values `LAMBDA` = 0.0, 1.0, 4.0, 50.0.
3. For each value of LAMBDA, simply call `stiff_direction()` and study the direction field plot.
4. It is possible to write this exercise in 3 lines of code!

You can see that the solution curve (red) is following a sequence of direction arrows. However, as `LAMBDA` increases, the behavior of the nearby direction vectors changes. When you reach `LAMBDA` = 4, no matter where you start in the rectangle, you head *very rapidly* (long arrows) toward $\sin x$, and then follow that curve as it varies *slowly* (short arrows). At `LAMBDA=50`, the direction vectors seem to point directly up or down, depending on which side of the solution curve they are.

Suppose that $LAMBDA = 50$, and we are solving the ODE with a new initial condition that is somewhere above the red solution curve. The picture tells us that in order to follow this solution, we need to choose a stepsize and move almost (but not exactly) downwards. The goal would be to come close to the red solution curve. But if we pick a stepsize that is too large, we will instead hop over the curve to the "up" side of the direction field. Our approximate solution may end up jumping back and forth, and maybe even growing in size. This phenomenon of overshooting and explosive growth is a characteristic of stiff differential equations; special ODE methods have been devised to try to deal with it.

11 Exercise 3

This exercise illustrates what is meant by stiffness of an ODE.

To begin with, let us estimate the work it ought to take to represent a solution of our stiff ODE. We are going to set `LAMDBA` to 10,000 and then plot the exact solution function over the interval $[0, 2\pi]$. Our question is, roughly now many evenly spaced x values do we need in order for the plot to look reasonable?

Your investigation might start out as follows, where you need to fill in the correct Python details:

```
some import statements
call stiff_lambda to set LAMBDA
choose n
set x to n equally spaced points in [0,2 pi]
set y to the solution of the stiff ODE
plot x and y
```

You should probably find that 40 points is about enough to show what the curve looks like.

Based on the plot, we might assume that we can compute a good approximation to the stiff ODE solution by using 40 equal steps in our ODE solver. Let's see what happens when we try this.

1. Create a file `exercise3.py` for this experiment.
2. Use `forward_euler.py` from the last lab.
3. Solve the stiff ODE, with `LAMBDA` set to 10,000, over the interval [0,2pi], with initial condition y(0) = 0, using `numSteps = 40`. Plot your results in red, and the exact solution in blue. Your results will be terrible.
4. Until you get reasonable results, multiply `numSteps` by 10 and try again.
5. Report the value of `numSteps` that you had to use in order to get an approximate solution whose plot looks close to the exact solution.

Our earlier plotting investigation suggested that the curve wasn't hard to draw, but our ODE solution efforts showed that the curve is actually very hard to follow!

12 The Backward Euler Method

The Backward Euler method is an important variation of Euler's method. Before we say anything more about it, let's take a hard look at the algorithm:

$$\begin{aligned} x_{k+1} &= x_k + h \\ y_{k+1} &= y_k + h f_{\text{ode}}(x_{k+1}, y_{k+1}) \end{aligned} \tag{2}$$

You might think there is no difference between this method and Euler's method. But look carefully–this is *not* a "recipe," the way some formulas are. Since y_{k+1} appears both on the left side and the right side, it is an equation that must be solved for y_{k+1}, *i.e.*, the equation defining y_{k+1} is implicit. It turns out that implicit methods are much better suited to stiff ODE's than explicit methods.

If we plan to use Backward Euler to solve our stiff ode equation, we need to address the method of solution of the implicit equation that arises. Before addressing this issue in general, we can treat the special case:

$$\begin{aligned} x_{k+1} &= x_k + h \\ y_{k+1} &= y_k + h\lambda(-y_{k+1} + \sin x_{k+1}) \end{aligned} \tag{3}$$

This equation can be solved for y_{k+1} easily enough! The solution is

$$\begin{aligned} x_{k+1} &= x_k + h \\ y_{k+1} &= (y_k + h\lambda \sin x_{k+1})/(1 + h\lambda) \end{aligned}$$

Now we will write a version of the backward Euler method that implements this solution, and only this solution. Later, we will look at more general cases.

13 back_euler_lam()

Create a file `back_euler_lam.py` with the signature:

```
def back_euler_lam ( LAMBDA, xRange, yInitial, numSteps )

  import numpy as np

  x = np.zeros ( numSteps + 1 )
  y = np.zeros ( ( numSteps + 1, np.size ( yInitial ) ) )

  dx = ( xRange[1] - xRange[0] ) / numSteps

  for k in range ( 0, numSteps + 1 ):

    if ( k == 0 ):
      x[0] = xRange[0]
      y[0,:] = yInitial
    else:
      x[k] = x[k-1] + dx
      y[k,:] = ???

  return x, y
```

14 Exercise 4

Now let's try again to solve our very stiff ODE, using the backward Euler method.

1. Create a file `exercise4.py` for this experiment.
2. Solving the system (**??**) with $\lambda = 10000$ over the interval $[0, 2\pi]$, starting from the initial condition $y = 0$ at `x=0` for 40 steps. Your solution should not blow up and its plot should look reasonable.
3. For the purpose of checking your work, the first few values of y are y=[0.0; 0.156334939127; 0.308919855800; 0.453898203657; ...]. Did you get these values?

Now we will compare backward Euler with forward Euler for accuracy. Of course, backward Euler gives results when the stepsize is large and Euler does not, but we are curious about the case that there are enough steps to get answers. Because it would require too many steps to run this test with $\lambda =$`1.e4`, you will be using $\lambda =$`55`.

15 Exercise 5

Now we will investigate the accuracy of the forward Euler method when applied to our stiff ODE. For this experiment, we will use a smaller value of `LAMBDA`, and we will focus only on the value we compute at the endpoint $x = 2\pi$. We want to see how the error in this quantity is reduced as we increase `numSteps`.

1. Create a file called `exercise5.py` for this experiment.
2. Use `stiff_lambda()` to set `LAMBDA` to 55.
3. Compute `yexact = stiff_ode ( 2.0 * np.pi )`;
4. Let numSteps = np.array ([40, 80, 160, 320, 640, 1280, 2560]);
5. For each `k`, solve the stiff ODE with `forward_euler()`, using numSteps[k].
6. For each `k`, Compute the final error e[k] = y[numSteps[k]] - yexact.
7. For all but the last `k`, compute r[k] = e[k] / e[k+1].
8. Based on the ratios in the table, estimate the order of accuracy of the method, *i.e.,* estimate the exponent p in the error estimate Ch^p. p is an integer in this case.

Although the results are very bad for the first few calculations, they should settle down.

16 Exercise 6

Now let's compare the backward Euler method.

1. Create a file called `exercise6.py` for this experiment.
2. Repeat the previous exercise, but now use `back_euler_lam()`.

Comment on the orders of accuracy of the forward and backward Euler methods.

17 Solving a nonlinear equation

You should be convinced that implicit methods are worth while. How, then, can the resulting implicit equation (usually it is nonlinear) be solved? We could build a backward Euler solver that applies Newton's method. This would require computing and supplying the partial derivative of f_{ode} with respect to y, and accessing a Newton nonlinear equation solver (which we did in an earlier lab).

However, there are already powerful tools available to us, in particular the function ttscipy.optimize.fsolve(), which can find the root of a nonlinear equation of the form $f(x) = 0$. All we have to do is supply `x0`, an initial guess for the root, and issue the command

```
from scipy.optimize import fsolve
x = fsolve ( f, x0 )
```

Our particular problem is a little more complicated. We need to explain to `fsolve()` that our problem doesn't have the simple form `f(x)=0`. Starting from the implicit equation

```
y[k] = y[k-1] + ( x[k] - x[k-1] ) * f ( x[k], y[k] )
```

we can think of our Backward Euler Function (bef) as

```
  bef = y[k] - y[k-1] - ( x[k] - x[k-1] ) * f ( x[k], y[k] )
```

and we need to ask `fsolve()` to find a `y[k]` that makes this zero.

Our `bef()` function has extra arguments, but here's how we call `fsolve()` with that extra information included:

```
  y[k,:] = fsolve ( bef, y[k,:], args = ( f_ode, x[k-1], y[k-1,:], x[k] ) )
```

Read this carefully. We are asking `fsolve()` to solve `bef=0` for a value `y[k,:]`, assuming that the ODE right hand side is `f_ode()`, the old data is `x[k-1], y[k-1,:]` and the new value of `x` is `x[k]`.

You don't need to worry about how I figured out this crazy formula, but you do need to understand what it is asking!

This means that, essentially, to get the backward Euler solver, we replace `just one line` in the forward Euler code, and add a function to evaluate bef.

18 back_euler()

1. Create `back_euler.py` by copying `forward_euler.py`.
2. Insert the statement `from scipy.optimize import fsolve`;
3. Replace the line defining `y[k,:]  =` by the line that calls fsolve.
4. Append the following definition of `bef()` into your code:

```
def bef ( yp, f, to, yo, tp ):
  value = yp - yo - ( tp - to ) * f ( tp, yp );
  return value
```

Test `back_euler()` on a simple version of the stiff ODE, by setting LAMBDA to 1, just to make sure it is working. We will give it a tougher test in the next exercise.

19 Exercise 7

1. Create a file called `exercise7.py` for this experiment.
2. Use `back_euler_lam()` to solve the stiff ODE, with `LAMBDA = 10000`, on the interval $[0, 2\pi]$, with initial value 0, for 40 steps, and save your solution as `y1[]`;
3. Use `back_euler()` to solve the same system, and save your solution as `y2[]`;
4. Print `x[k], y1[k], y2[k]` for $0 \leq k < numSteps + 1$. The two solutions should agree to ten or more significant digits.

Assuming your solutions agree, we can expect that `back_euler()` is working properly.

20 van der Pol's equation

One simple nonlinear equation with quite complicated behavior is van der Pol's equation. This equation is written

$$z'' + \nu(z^2 - 1)z' + z = e^{-x}$$

where e^{-x} has been chosen as a forcing term that dies out as x gets large. When $z > 1$, this equation behaves somewhat as an oscillator with negative feedback ("damped" or "stable"), but when z is small then it looks like an oscillator with positive feedback ("negatively damped" or "positive feedback" or "unstable"). When z is small, it grows. When z is large, it dies off. The result is a non-linear oscillator. Physical systems that behave somewhat as a van der Pol oscillator include electrical circuits with semiconductor devices such as tunnel diodes in them and some biological systems, such as the beating of a heart, or the firing of neurons. In fact, van der Pol's original paper was, "The Heartbeat considered as a Relaxation oscillation, and an Electrical Model of the Heart," Balth. van der Pol and J van der Mark, *Phil. Mag. Suppl.* 6(1928) pp 763—775. More information is available in an interesting Scholarpedia article.

As you know, van der Pol's equation can be written as a system of first-order ODEs in the following way.

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2) = y_2 \\ y_2' &= f_2(x, y_1, y_2) = -\nu(y_1^2 - 1)y_2 - y_1 + e^{-x} \end{aligned} \tag{4}$$

where $z = y_1$ and $z' = y_2$ and ν is a parameter.

21 vanderpol_ode()

The van der Pol equation is a good test for our backward Euler ODE solver; it is a system rather than a single equation, it's stiff, and it's a nonlinear ODE, which means that, unlike with the case of our "stiff_ode", we cannot rearrange the algorithm to make it a simple explicit formula for the next values.

Like our stiff ODE, the van der Pol equation includes a parameter ν. We will assign this quantity a default value of 11.0 initially, but later we may want to vary it. For that reason, we will want to write a function `vanderpol_nu()` that allows us to easily set or get this parameter value.

Create a file `vanderpol_ode.py`, and start out by implementing the right hand side (which will be a vector this time) in the usual way. The signature will be:

```
def vanderpol_ode ( x, y ):

  NU = vanderpol_nu ( )
  fValue = np.zeros ( 2 )
  fValue[0] = ???
  fValue[1] = ???
  return fValue
```

Replace the question marks by the appropriate formulas. Then append the following function definition to allow us to set or get the current value of `NU` also known as ν.

```
def vanderpol_nu ( nu_input = None ):

  if ( not hasattr ( vanderpol_nu, "nu_default" ) ):
    vanderpol_nu.nu_default = 11.0

  if ( nu_input is not None ):
    vanderpol_nu.nu_default = nu_input

  nu_output = vanderpol_nu.nu_default

  return nu_output
```

You could prepare for a test run of your code with the commands:

```
from vanderpol_ode import vanderpol_ode, vanderpol_nu
NU = 11.0     # Choose a value for NU
vanderpol_nu ( NU )  # Notify all codes of the value of NU
```

then set the appropriate input and call `forward_euler` or back_euler() and plot the results. We will try this in the next exercise.

22 Exercise 8

1. Create a file called `exercise8.py` for this experiment.
2. Set `NU` to 11.
3. Using `forward_euler()`, solve the van der Pol system over the interval from `x=0` to `x=2`, starting from `y=[0;0]` and using 40 steps. Save the solution values as `x1, y1`.
4. Using the same data, solve the system with `back_euler()`. Save the solution values as `x2, y2`.
5. Show both solutions on one plot. You should see that they don't agree, and that the solution produced by forward_euler() seems less likely to be correct.

You may be interested to see what happens if you rerun your program with a larger value of `NU`. Even for a value of `NU=12`, the forward Euler code will deteriorate, and for larger values, the solution is worthless unless we increase the number of steps.

23 The Midpoint Method

The implicit midpoint method for solving an ODE tries to improve the accuracy of the predicted value y_k. It does this by using a backward Euler approximation with a half step in x, producing the pair `(xm,ym)`. We initialize this process by:

$$\begin{aligned} xm &= x_{k-1} + dx/2 \\ ym &= y_{k-1} + dx/2\, f(x_{k-1}, y_{k-1}) \quad \text{(starting estimate for ym)} \end{aligned}$$

and then we call fsolve() to solve the implicit equation.

$$ym = y_{k-1} + dx/2\, f(xm, ym) \quad \text{Implicit equation! Need to call fsolve()}$$

Previously, in `back_euler()`, the call to fsolve() looked like this:

```
y[k,:] = fsolve ( bef, y[k,:], args = ( f_ode, x[k-1], y[k-1,:], x[k] ) )
```

We will have to modify the call to `fsolve()`, so that the values of `xm` and `ym` replace the occurrence of `x[k]` and the 2 occurrences of `y[k,:]`.

Once we have computed `ym`, then y_k is approximated by

$$y_k = 2\, ym - y_{k-1}$$

Compare to the backward Euler method, the midpoint method seems to involve a bit more work, so we can hope that there will be some improvement in performance!

24 midpoint()

1. Create `midpoint.py` by copying `back_euler.py`.
2. Before the call to fsolve(), insert the two lines that define `xm` and initialize the estimate for `ym`:

```
xm = x[k-1] + ?
ym = y[k-1,:] + ?
```

3. Modify the call to `fsolve()` so that it uses `xm` and `ym` in place of `x[k-1]` and `y[k-1,:]`;
4. As suggested above, write the formula that evaluates `y[k,:]` based on `y[k-1,:]` and `ym`

```
y[k,:] = ?
```

Test `midpoint()` on a simple version of the stiff ODE, by setting `LAMBDA` to 1, just to make sure it is working. We will give it a tougher test in the next exercise.

25 Exercise 9

To make sure that your midpoint rule has been implemented correctly, let's try it out on the same problem we looked at in exercise 8.

1. Create a file called `exercise9.py` for this experiment.
2. Set `NU` to 11.
3. Using `midpoint()`, solve the van der Pol system over the interval from `x=0` to `x=2`, starting from `y=[0;0]` and using 40 steps.
4. Plot the solution.

How does your plot compare to the results from exercise 8?

26 Exercise 10

In exercises 5 and 6, you ran `forward_euler` and `back_euler_lam` on the stiff ODE, for an increasing sequence of number of steps. This allowed you to compute the error, and then observe the error ratios, which suggested the rate of convergence of the method. We will now try this same test for the midpoint method. You might save some time by starting from a copy of `exercise5.py`

1. Create a file called `exercise10.py` for this experiment.
2. Use `stiff_lambda()` to set `LAMBDA` to 55.
3. Compute `yexact = stiff_ode ( 2.0 * np.pi );`
4. Let `numSteps = np.array ( [ 40, 80, 160, 320, 640, 1280, 2560] );`
5. For each `k`, solve the stiff ODE with `midpoint()`, using numSteps[k].
6. For each `k`, Compute the final error`e[k] = y[numSteps[k]] - yexact`.
7. For all but the last `k`, compute `r[k] = e[k] / e[k+1]`.
8. Based on the ratios in the table, estimate the order of accuracy of the method, the exponent p in the error estimate Ch^p. p is an integer in this case.