Lab 2: Explicit ODE methods
MATH2071, University of Pittsburgh, Spring 2023

1 Introduction

In this lab we consider solution methods for ordinary differential equations (ODEs). We will be looking at two classes of methods that excel when the equations are smooth and derivatives are not too large. This lab will take two class sessions.

The lab begins with an introduction to Euler's (explicit) method for ODEs. Euler's method is the simplest approach to computing a numerical solution of an initial value problem. However, it has about the lowest possible accuracy. If we wish to compute very accurate solutions, or solutions that are accurate over a long interval, then Euler's method requires a large number of small steps. Since most of our problems seem to be computed "instantly," you may not realize what a problem this can become when solving a "real" differential equation.

Applications of ODEs are divided between ones with space as the independent variable and ones with time as the independent variable. We will use x as independent variable consistently. Sometimes it will be interpreted as a space variable (x-axis) and sometimes as time.

A number of methods have been developed in the effort to get solutions that are more accurate, less expensive, or more resistant to instabilities in the problem data. Typically, these methods belong to "families" of increasing order of accuracy, with Euler's method (or some relative) often being the member of the lowest order.

In this lab, we will look at "explicit" methods, that is, methods defined by an explicit formula for y_{k+1}, the approximate solution at the next time step, in terms of quantities derivable from previous time step data. In a later lab, we will address "implicit" methods that require the solution of an equation in order to find y_{k+1}. We will consider the Runge-Kutta and the Adams-Bashforth families of methods. We will talk about some of the problems of implementing the higher order versions of these methods. We will try to compare the accuracy of different methods applied to the same problem, and using the same number of steps.

Runge-Kutta methods are "single-step" methods while Adams-Bashforth methods are "multistep" methods. Multistep methods require information from several preceding steps in order to find y_{k+1} and are a little more difficult to use. Nonetheless, both single and multistep methods have been very successful and there are very reliable routines available to solve ODEs using both types of methods.

2 Euler's method

A very simple ordinary differential equation (ODE) is the *explicit scalar first-order initial value problem*:

$$\begin{aligned} \frac{dy}{dx} &= f_{\text{ode}}(x,y) \\ y(x_0) &= y_0. \end{aligned}$$

The equation is *explicit* because dy/dx can be written explicitly as a function of x and y. It is *scalar* because we assume that $y(x)$ is a scalar quantity, not a vector. It is *first-order* because the highest derivative that appears is the first derivative dy/dx. It is an *initial value problem* (IVP) because we are given the value of the solution at some time or location x_0 and are asked to produce a formula for the solution at later times.

An *analytic solution* of an ODE is a formula $y(x)$, that we can evaluate, differentiate, or analyze in any way we want. Analytic solutions can only be determined for a small class of ODE's. The term "analytic" used here is not quite the same as an analytic function in complex analysis.

A "numerical solution" of an ODE is simply a list of abscissas x and values y arranged in table (x_k, y_k) that approximate the value of an analytic solution $y(x)$. A numerical solution is only an approximation at a discrete set of points; there is generally some difference between the tabulated values and the true solution.

The important question is, how large is this difference, and how does it grow as we proceed? One way to pose this question is to determine how close the computed values (x_k, y_k) are to the analytic solution, which we might write as $(x_k, y(x_k))$.

The simplest method for producing a numerical solution of an ODE is known as *Euler's explicit method*, or the *forward Euler method*. Given a solution value (x_k, y_k), we estimate the next row of the approximate solution table by:

$$\begin{aligned} x_{k+1} &= x_k + h \\ y_{k+1} &= y_k + h\, y'(x_k, y_k). \end{aligned}$$

(The step size is denoted h here. Sometimes it is denoted dx.) We can take as many steps as we want with this method, using the result from one step as the starting point for the next step.

A procedure to solve an ODE will need to accept as an argument the name of a function which evaluates y'. That function can have any name, but the procedure gives it a "dummy name", which we will usually take to be "`f_ode(x,y)`". Inside of the procedure, we can evaluate the function as though that was its actual name.

An ODE solving procedure might have a signature like this:

```
def ode_solver ( f_ode, xRange, yInitial, numSteps ):
...
  return x, y
```

Supposing your right hand side function is evaluated by `my_ode(x,y)`, then you call `ode_solver()` using a command such as

```
f_ode = my_ode
xRange = np.array ( [ 0.0, 1.0 ] )
yInitial = 0.0
numSteps = 10
x, y = ode_solver ( f_ode, xRange, yInitial, numSteps )
```

or perhaps simply

```
x, y = ode_solver ( my_ode, np.array ( [0.0,1.0] ), 0.0, 10 )
```

3 forward_euler()

Our first example of an ODE solver implements Euler's method, and is called `forward_euler()`. It is written in such a way that it will be able to handle systems of ODE's as well as a single ODE. Download the file `forward_euler.py` or make a copy from the text here using cut-and-paste:

```
def forward_euler ( f_ode, xRange, yInitial, numSteps ):
  """
  x, y = forward_euler ( f_ode, xRange, yInitial, numSteps )

  Use Euler's explicit method on one or more ODEs dy/dx=f_ode(x,y).

  Input:
    f_ode evaluates the right hand side.
    xRange = [x1,x2], the solution interval.
    yInitial = k initial values for y at x1
    numSteps = number of equally-sized steps to take from x1 to x2
  Output:
    x = numSteps+1 x values.
    y = numSteps+1 rows and k columns, with k-th row containing solution at x[k]
  """
  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,:] = y[k-1,:] + dx * f_ode( x[k-1], y[k-1,:] )

  return x, y
```

4 expm_ode, a sample ODE problem

Here is a sample ODE to be solved:

$$\begin{aligned} \frac{dy}{dx} &= -y - 3x \\ y(0) &= 1 \end{aligned} \tag{1}$$

The exact solution is $y = -2e^{-x} - 3x + 3$.

Create a file `expm_ode.py` defining the right hand side of this problem:

```
def expm_ode ( x, y ):
  """
  expm_ode ( x, y ) evaluates the ODE right hand side dy/dx=-y+3*x

  Input:
    x is the independent variable
    y is the dependent variable
  Output:
    fValue is the value of dy/dx
  """
  fValue = - y - 3.0 * x

  return fValue
```

5 Exercise 1

1. Create a file `exercise1.py` for this exercise.
2. Use `forward_euler()` to solve the sample problem,

```
f_ode = expm_ode
xRange = np.array ( [ 0.0, 2.0 ] )
yInit = 1.0
numSteps = ?
x, y = forward_euler ( f_ode, xRange, yInit, numSteps )
```

   for each of the 6 values of `numSteps` in the table below.
3. For `k` from 0 to 5, using `numSteps[k]`, compute the error `e[k]` as the difference between your approximate solution and the exact solution at the final point `x=2.0`.
4. for `k` from 0 to 4, compute and print the error ratios `r[k] = e[k] / e[k+1]`.

```
                          Euler's explicit method
                          Value         Error        Error ratio
k numSteps Stepsize       Y[-1,0]       E[k]         E[k]/E[k+1]

0    10     0.2      -3.21474836   5.5922e-02   _________
1    20     0.1       __________   __________   _________
2    40     0.05      __________   __________   _________
3    80     0.025     __________   __________   _________
4   160     0.0125    __________   __________   _________
%   320     0.00625   __________   __________
```

5. You know the error is $O(h^p)$ for some p. There is a simple way to estimate the value of p by successively halving h. If the error were *exactly* Ch^p, then by solving twice, once using h and the second time using $h/2$ and taking the ratio of the errors, you would get

$$\frac{\text{error}(h)}{\text{error}(h/2)} = \frac{Ch^p}{C(h/2)^p} = 2^p.$$

Since the error is only $O(h^p)$, the ratio is only approximately 2^p.
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, where h is the step size. p is an integer in this case.

6 The Euler Halfstep (RK2) Method

The "Euler halfstep" or "RK2" method is a variation of Euler's method. It is the second-simplest of a family of methods called "Runge-Kutta" methods. As part of each step of the method, an auxiliary solution, one that we don't really care about, is computed halfway, using Euler's method:

$$\begin{aligned} x_a &= x_k + h/2 \\ y_a &= y_k + 0.5hf_{\text{ode}}(x_k, y_k) \end{aligned} \tag{2}$$

The derivative function is evaluated at this point, and used to take a full step from the original point:

$$\begin{aligned} x_{k+1} &= x_k + h; \\ y_{k+1} &= y_k + hf_{\text{ode}}(x_a, y_a) \end{aligned} \tag{3}$$

Although this method uses Euler's method, it ends up having a higher order of convergence. Loosely speaking, the initial half-step provides additional information: an estimate of the derivative in the middle of the next step. This estimate is presumably a better estimate of the overall derivative than the value at the left end point. The per-step error is $O(h^3)$ and, since there are $O(1/h)$ steps to reach the end of the range, $O(h^2)$ overall. Keep in mind that we do not regard the auxiliary points as being part of the solution. We throw them away, and make no claim about their accuracy. It is only the whole-step points that we want.

7 rk2()

Write a file named `rk2.m` that implements the Euler halfstep (RK2) method sketched above in Equations (**??**) and (**??**). Keep the same calling parameters and results as for `forward_euler.m` above. Keeping these the same will make it easy to compare different methods. The following model for the file is based on the `forward_euler.m` file with the addition of the variables `xa` and `ya` representing the auxiliary variables x_a and y_a in Equation (**??**). Add comments to this outline, including explanations of all the variables in the signature line, and fill in expressions where `???` have been left.

```
def rk2 ( f_ode , xRange, yInitial , numSteps ):
  """
  x, y = rk2 ( f_ode , 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:
      xa = ???
      ya = ???
      x[k] = x[k-1] + dx
      y[k,:] = y[k-1,:] + dx * f_ode( ??? )

  return x, y
```

Notice that the output quantity `y` is treated like a “matrix”. This allows us to use the same ODE solver for scalar problems and vector problems of any size. However, this means that if we are actually solving a scalar problem, we still have to think of the result `y` as a matrix, not a vector; if the last entry in `y` is the solution value at the end point, we have to access this value using the expression `y[-1,0]`, where “-1” means *last row* and “0” means *first* (and in this case, only) item in the row. If we were to write `y[-1]`, we wouldn’t get a number, we’d get a numpy() array containing that number, which is not what you want!

8 Exercise 2

1. Write a file `exercise2.py` for this exercise.
2. Use `rk2()` to compute the numerical solution of the model ODE for the exponential, `expm_ode.m`, from Exercise 1, from `x = 0.0` to `x = 2.0`, and with the same initial value as in Exercise 1, but using Euler’s halfstep method, RK2, with stepsizes below.
3. For each case, record the value of the numerical solution at `x = 2.0`; the error, that is, the difference between the numerical solution and the true solution at the end point `x=2` (`y=-2*exp(-2)-3`); and, the ratios of the error for each value of `numSteps` divided by the error for the succeeding value of `numSteps`.

`Using RK2 Solver:`

			Value	Error	Error Ratio
k	numSteps	Stepsize	Y[-1,0]	E[K]	E[K]/E[K+1]
0	10	0.2	-3.274896063	4.2255e-3	__________
1	20	0.1	__________	__________	__________
2	40	0.05	__________	__________	__________
3	80	0.025	__________	__________	__________
4	160	0.0125	__________	__________	__________
5	320	0.00625	__________	__________	

4. Based on the ratios in the table, estimate the order of accuracy of the method, that is, estimate the exponent p in the error estimate Ch^p. p is an integer in this case.

5. Compare errors from Euler's method (Exercise 1) and Euler's halfstep method for this problem. You should clearly see that Euler's halfstep method (RK2) converges much faster than Euler's method.

			Euler	RK2
k	numSteps	Stepsize	Error E[K]	Error E[k]
0	10	0.2	----------	----------
1	20	0.1	----------	----------
2	40	0.05	----------	----------
3	80	0.025	----------	----------
4	160	0.0125	----------	----------
5	320	0.00625	----------	----------

6. Based on the above table, roughly how many steps does Euler require to achieve the accuracy that RK2 has for `numSteps=10`?
7. You have already found the the error for Euler's method is approximately $C_{\mathrm{E}}h^{p_{\mathrm{E}}}$ and the error for RK2 is approximately $C_{\mathrm{RK2}}h^{p_{\mathrm{RK2}}}$. Based on one or both of these estimates, roughly how many steps would Euler require to achieve the accuracy that RK2 has for `numSteps=320`? Explain your reasoning.
8. Check that the accuracy obtained using Euler's method with your estimated number of steps is comparable to the accuracy that RK2 has for `numSteps=320`.

9 Runge-Kutta Methods

The idea in Euler's halfstep method is to "sample the water" between where we are and where we are going. This gives us a much better idea of what f is doing, and where our new value of y ought to be. Euler's method ("RK1") and Euler's halfstep method ("RK2") are the junior members of a family of ODE solving methods known as "Runge-Kutta" methods.

To develop a higher order Runge-Kutta method, we sample the derivative function f at even more "auxiliary points" between our last computed solution and the next one. These points are *not* considered part of the solution curve; they are just a computational aid. The formulas tend to get complicated, but let me describe the next one, at least.

The third order Runge Kutta method "RK3," given x, y and a stepsize h, computes two intermediate points:

$$\begin{aligned} x_a &= x_k + .5h \\ y_a &= y_k + .5h f_{\mathrm{ode}}(x_k, y_k) \\ x_b &= x_k + h \\ y_b &= y_k + h(2f_{\mathrm{ode}}(x_a, y_a) - f_{\mathrm{ode}}(x_k, y_k)) \end{aligned} \tag{4}$$

and then estimates the solution as:

$$\begin{aligned} x_{k+1} &= x_k + h \\ y_{k+1} &= y_k + h(f_{\mathrm{ode}}(x_k, y_k) + 4.0f_{\mathrm{ode}}(x_a, y_a) + f_{\mathrm{ode}}(x_b, y_b))/6.0 \end{aligned} \tag{5}$$

The global accuracy of this method is $O(h^3)$, and so we say it has "order" 3. Higher order Runge-Kutta methods have been computed, with those of order 4 and 5 the most popular.

10 rk3()

Write a file called `rk3.m` with the signature

```
function [ x, y ] = rk3 ( f_ode, xRange, yInitial, numSteps )
% comments including the signature, meanings of variables,
% math methods, your name and the date
```

that implements the above algorithm. You can use `rk2.m` as a model.

11 Exercise 3

1. Write a file `exercise3.py` for this exercise.
2. Using `rk3()`, repeat the numerical experiment in Exercise 2 (using `expm_ode`) and fill in the following table. Use the first line of the table to confirm that you have written the code correctly.

		RK3	RK3	
numSteps	Stepsize	Y[-1,0]	Error E[K]	Ratio=E[K]/E[K+1]
10	0.2	-3.27045877	2.1179e-04	__________
20	0.1	__________	__________	__________
40	0.05	__________	__________	__________
80	0.025	__________	__________	__________
160	0.0125	__________	__________	__________
320	0.00625	__________	__________	

3. 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.
4. Compare errors from the RK2 method (Exercise 2) and the RK3 method for this problem. You should clearly see that RK3 converges much faster than RK2.

		RK2	RK3
numSteps	Stepsize	Error E[K]	Error E[K]
10	0.2	__________	__________
20	0.1	__________	__________
40	0.05	__________	__________
80	0.025	__________	__________
160	0.0125	__________	__________
320	0.00625	__________	__________

5. Based on the above table, roughly how many steps does RK2 require to achieve the accuracy that RK3 has for `numSteps=10`?
6. You have already found the the error for RK2 is approximately $C_{\text{RK2}}h^{p_{\text{RK2}}}$ and the error for RK3 is approximately $C_{\text{RK3}}h^{p_{\text{RK3}}}$. Based on one or both of these estimates, roughly how many steps would RK2 require to achieve the accuracy that RK3 has for `numSteps=320`? Explain your reasoning.
7. Check that the accuracy obtained using RK2 with your estimated number of steps is comparable to the accuracy that RK3 has for `numSteps=320`.

12 Exercise 4

You have not tested your code for *systems* of equations yet. In this exercise you will do so by solving the "system"

$$\begin{aligned}\frac{dy_0}{dx} &= -y_0 - 3x \\ \frac{dy_1}{dx} &= -y_1 - 3x. \end{aligned} \tag{6}$$

You can see that this "system" is really (**??**) twice, so you can check y_0 and y_1 against each other and against your earlier work.

1. Write a file `exercise3.py` for this exercise.
2. The file `expm_ode.m` will automatically return a vector result if it is given a vector for `y`. To see this fact in action, try the following command:

   ```
   x = 1.0
   y = np.array ( [ 5, 6 ] )
   fValue = expm_ode ( x, y )
   print ( fValue )
   ```

   Please include the value of `fValue` in your summary file.
3. Solve the system (**??**) using `rk3` and `expm_ode` on the interval [0,2] starting from the initial value *vector* [5,6], with `numSteps=40`. Save the results as `xs, ys`. The result at the final time `xs[-1]` is stored in `ys[-1,:]`. Print out this value.
   (Recall that, in Python, asking for the -1 index of an array asks for the `last` item in that dimension. Using an index array of : asks for `all` the items in that dimension. We are interested in the last row of `ys`, and all the entries in that row.)
4. Solving the system (**??**) amounts to solving (**??**) twice, once with initial value y(0)=5, and once more with initial value y(0)=6. To verify this, repeat the calculation by solving the scalar IVP (**??**), once with initial value `y=5` and once with initial value `y=6`. Save these solutions as `x0, y0` and `x1, y1`. Compare `y0[-1]` and `ys[-1,0]`. Compare `y1[-1]` and `ys[-1,1]`. These values should match.
5. Instead of comparing the last entries of the two calculations, you can get a measure of the difference of all the entries, using the `norm()` function:

   ```
   import numpy as np
   np.linalg.norm ( ys[:,0] - y0[:,0] )
   np.linalg.norm ( ys[:,1] - y1[:,0] )
   ```

   Note that, even though `y0` and `y1` are "really" vectors, the ODE solver returns each of them as a "matrix", that is, an object that must be accessed by two indices. So, for instance, instead of writing `y0[:]` to represent the list of solution values of the first scalar ODE, we have to write `y0[:,0]`.

13 pendulum_ode()

The equation describing the motion of a pendulum can be described by the single dependent variable θ representing the angle the pendulum makes with the vertical. The coefficients of the equation depend on the length of the pendulum, the mass of the bob, and the gravitational constant. Assuming a coefficient value of 3, the equation is

$$\frac{d^2\theta}{dx^2} + 3\sin\theta = 0$$

and one possible set of initial conditions is

$$\begin{aligned} \theta(x_0) &= 1 \\ \frac{d\theta}{dx}(x_0) &= 0 \end{aligned}$$

This second order equation can be written as a system

$$\begin{aligned} \frac{dy}{dx} &= \begin{pmatrix} y_2 \\ -3\sin y_1 \end{pmatrix} \\ y(0) &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}. \end{aligned}$$

(Recall that this transformation is accomplished by the change of variables $y_0 = \theta$ and $y_1 = d\theta/dx$.)

Write a file named `pendulum_ode.py` with signature

```
def pendulum_ode ( x, y ):
  """
  fValue = pendulum_ode(x,y)
  comments including meanings of variables,
  math methods, your name and the date
  """
...
  return fValue
```

14 Exercise 5

1. Write a file `exercise5.py` for this exercise.
2. Generate a solution (x1,y1) using Euler with 1,000 steps.
3. Generate a solution (x2,y2) using Euler with 10,000 steps.
4. Generate a solution (x3,y3) using RK3 with 100 steps.
5. Make three plots of the theta portion of each of your solutions, using commands like:

   ```
   import matplotlib.pyplot as plt
   theta1 = y1[:,0]
   plt.plot ( x1, theta1 )
   ```

6. Conservation of energy guarantees that physically, the value of θ should stay between -1 and 1. What do your plots suggest about the accuracy of your results?

15 Stability

Explicit methods for solving ODE's are generally "conditionally stable". In order for the computed solution to approximate the true solution well, the stepsize `h` must be small enough (or equivalent, the value of `numSteps` must be large enough.)

In the following exercise, we will start with a large value of `numsteps` and decrease it repeatedly. As the stepsize `h` grows, not only do the results become less accurate, but the solution will become unstable, resulting in a crazy see-saw plot.

16 Exercise 6

1. Write a file `exercise6.py` for this exercise.
2. Use `forward_euler` to solve the ODE using `expm_ode`, over the interval [0,20], starting from `y=20` and using `numSteps=40, 30, 20, 15, 12, 10`. As you compute each solution, add it to a single plot. Once all your solutions have been plotted, you could add a `legend()` command to identify the individual curves:

   ```
     plt.legend ( [ 'n=40', 'n=30', 'n=20', 'n=15', 'n=12', 'n=10' ] )
   ```

3. Use `RK3` to solve the ODE using `expm_ode`, over the interval [0,20], starting from `y=20` and using `numSteps=40, 30, 20, 15, 12, 10, 8`. Note that this list has one more value than we used for the Euler example. As you compute each solution, add it to a single plot.

In both your Euler and RK3 plots, you should see that, as the stepsize becomes too large, the approximate solution starts jumping up and down. This is a sign that the ODE procedure has become unstable, and that the stepsize is too large to get accurate results.

17 Adams-Bashforth Methods

Like Runge-Kutta methods, Adams-Bashforth methods want to estimate the behavior of the solution curve, but instead of evaluating the derivative function at new points close to the next solution value, they look at the derivative at old solution values and use interpolation ideas, along with the current solution and derivative, to estimate the new solution. This way they don't compute solutions at auxiliary points and then throw the auxiliary values away. The savings can result in increased efficiency.

Looked at in this way, the forward Euler method is the first order Adams-Bashforth method, using *no* old points at all, just the current solution and derivative. The second order method, which we'll call "AB2," adds the derivative at the previous point into the interpolation mix. We might write the formula this way:

$$y_{k+1} = y_k + h(3f_{\text{ode}}(x_k, y_k) - f_{\text{ode}}(x_{k-1}, y_{k-1}))/2$$

The AB2 method requires derivative values at two previous points, but we only have one when starting out. If we simply used an Euler step, we would pick up a relatively large error on the first step, which would pollute all subsequent results. In order to get a reasonable starting value, we should use the RK2 method, whose per-step error is order $O(h^3)$, the same as the AB2 method.

18 ab2()

The code `ab2()` implements the Adams-Bashforth second-order method. Download the file `ab2.py`, or else try using cut-and-paste on the text, so that you can use this function in the next exercise.

```
def ab2 ( f_ode, xRange, yInitial, numSteps ):
  """
  [ x, y ] = ab2 ( f_ode, xRange, yInitial, numSteps )

  uses Adams-Bashforth second-order method to solve a system
  of first-order ODEs dy/dx=f_ode(x,y).

  Input:
    f_ode = evaluates right hand side.
    xRange = [x1,x2] where the solution is sought on x1<=x<=x2
    yInitial = column vector of initial values for y at x1
    numSteps = number of equally-sized steps to take from x1 to x2
  Output:
    x = vector of values of x
    y = matrix whose k-th row is the approximate solution at x(k).
  """
  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[k] = xRange[0]
      y[k,:] = yInitial
    elif ( k == 1 ):
      fValue = f_ode ( x[k-1], y[k-1,:] )
      xhalf = x[k-1] + 0.5 * dx
      yhalf = y[k-1,:] + 0.5 * dx * fValue
      fValuehalf = f_ode ( xhalf, yhalf )
      x[k] = x[k-1] + dx
      y[k,:] = y[k-1,:] + dx * fValuehalf
    else:
      fValueold = fValue
      fValue = f_ode ( x[k-1], y[k-1,:] )
      x[k] = x[k-1] + dx
      y[k,:] = y[k-1,:] + dx * ( 3.0 * fValue - fValueold ) / 2.0

  return x, y
```

- The temporary variables `fValue` and `fValueold` have been introduced here but were not needed in the Euler, RK2 or RK3 methods. Explain, in a few sentences, their role in AB2.
- If `numSteps` is 100, then *exactly* how many times will we call the derivative function `f_ode`?

19 Exercise 7

1. Create a file `exercise7.py` for this exercise.
2. Use `ab2()` to compute the numerical solution of the ODE for the exponential (`expm_ode`) from `x = 0.0` to `x = 2.0`, starting at `y=1` with `numSteps` = 10, 20, 40, 80, 160, 320. Recall that the exact solution at `x=2`, is `y=-2*exp(-2)-3`. For each case, record the value of the numerical solution at `x = 2.0`, and the error

```
                AB2 errors on expm_ode

k numSteps  y[-1,0]    Error E[k]

0   10      -3.28013993  9.4694e-03
1   20       ----------  ----------
2   40       ----------  ----------
3   80       ----------  ----------
4  160       ---------   ----------
5  320       ---------   ----------
```

3. Make a second table reporting the error ratios:

```
                AB2 error ratios on expm_ode

k numSteps  Ratio E[k]/E[k+1]

0   10       ----------
1   20       ----------
2   40       ----------
3   80       ----------
4  160       ----------
```

4. Based on the ratios in the table, estimate the order of accuracy of AB2, that is, the exponent p in the error estimate Ch^p.