Numerical Solution of an Ordinary Differential Equation with 1 dependent variable

October 29, 2008

---

1 Statement of the problem

We wish to find solutions of a differential equation of the form

$$\frac{dy}{dx} = F(x, y) \tag{1}$$

where $y(x)$ is a real-valued function of one variable and $F(x, y)$ is a real valued function of two real variables giving the slope dy/dx at the point (x, y). The variable y is called the *dependent variable* and the variable x is called the *independent variable*. The labeling of the variables will vary with the problem. For example, in $dx/dt = -x^2$, x is the dependent variable and t (think "time") is the independent variable.

2 Description of the algorithm

The approximate solution starts by dividing up the x dimension into finite intervals of width Δx, so that $x_j = x_o + j\Delta x$. The object is to find the corresponding y_j, which is shorthand for $y(x_j)$. If we know the *mean* slope $s_j \equiv (y_{j+1} - y_j)/\Delta x$ in the jth interval, then the solution at the points x_j is given *exactly* by the formula

$$y_{j+1} = y_j + s_j \Delta x \tag{2}$$

which can be solved from left to right starting from knowledge of y_o. The procedure is illustrated in the left panel of Fig. 1. The rub is that we must find some way of evaluating the mean slope s_j.

The function $F(x, y)$ gives us the slope at a given point, but we can't use it to get the exact answer because we don't know $y(x)$ between the current value x_j and the next point we want to solve for. However, if F is a continuous function (think "well behaved, no jumps or other wild behavior") then when Δx is small the mean slope will be approximated by the slope evaluated at either

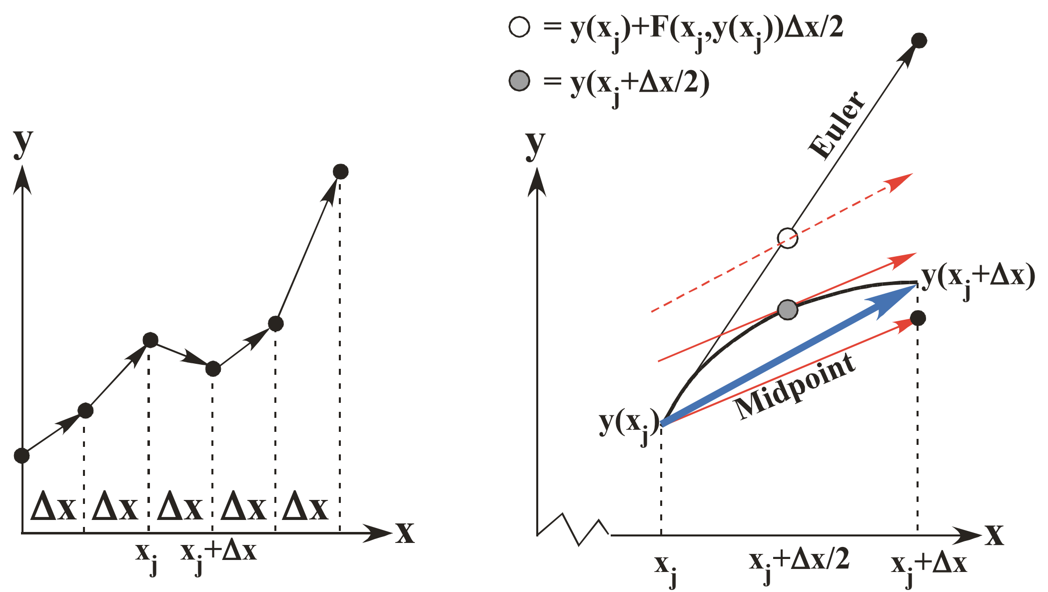

Figure 1:

endpoint of the interval. Thus, a simple approximation is to set $s_j = F(x_j, y_j)$, which is known if we know the solution up to the point (x_j, y_j). This leads to the iteration

$$y_{j+1} = y_j + F(x_j, y_j)\Delta x \tag{3}$$

which is known as *Euler's method*. It works, and it does give us an answer which converges to the correct solution as Δx is made smaller, but the error doesn't decrease very rapidly as Δx decreases. In fact, it can be shown that the error only decreases in proportion to Δx. If you want to improve the accuracy by a factor of 100, you need to use 100 times more points to get from the starting x to the finish line. The result of one Euler step, relative to the correct solution, is illustrated by the arrow marked "Euler" in the right hand panel of Fig. 1.

With a little more work we can do better. If we instead evaluate the slope function F at the *midpoint* of the interval, we will have a better approximation to the mean slope. The value we want is $F(x_j + \Delta x/2, y(x_j + \Delta x/2))$, and the resulting slope is indicated by the red solid arrow in Fig. 1. The problem is that we don't know $y(x_j + \Delta x/2))$, so we can't evaluate the function there. However, we can *approximate* this midpoint value of y by $y_m \equiv y_j + F(x_j, y_j)\Delta x/2$ and then set $s_j = F(x_j + \Delta x/2, y_m)$; this slope is indicated by the dashed red arrow in Fig. 1, and is a close approximation to the actual midpoint slope. Using this estimate yields the *midpoint method*, summarized by the iteration

$$\begin{aligned} y_m &= y_j + \frac{1}{2}F(x_j, y_j)\Delta x \\ y_{j+1} &= y_j + F(x_j + \Delta x/2, y_m)\Delta x \end{aligned} \tag{4}$$

With a fair amount of algebra it can be shown that the error in this case decreases in proportion to $(\Delta x)^2$. Now, to increase the accuracy by a factor of 100, we only need to use 10 times as many points.

With extreme cleverness and vastly many pages of algebra, it was discovered that there is a way of making use of evaluations of the slope function at *four* intermediate points in order to yield an algorithm whose error decreases in proportion to $(\Delta x)^4$. This is the *Runge-Kutta* algorithm. It is much harder to justify on a graphical basis,but the mathematics underpinning the accuracy of the method is firm. It is so subtle though, that there is no known way of formulating higher order extensions of the method involving more intermediate evaluations; indeed it is not even know whether such methods *exist* for arbitrarily high orders. The question is academic, however, since the fourth order Runge-Kutta method suffices for almost all scientific purposes. It is a kind of gift from the Gods of Mathematics that such a thing exists. The Runge-Kutta method is defined by the iteration

$$\begin{aligned} &hh = \Delta x/2, h6 = \Delta x/6, xh = x_j + hh, dydx = F(x_j, y_j) \\ &yt = y_j + hh \cdot dydx, dyt = F(xh, yt), yt1 = y + hh \cdot dyt \\ &dym = F(xh, yt1), yt2 = y + \Delta x \cdot dym, dym1 = dym + dyt, dyt1 = F(x_j + \Delta x, yt2) \\ &y_{j+1} = y_j + h6 \cdot (dydx + dyt1 + 2dym1) \end{aligned} \tag{5}$$

3 Performance of the algorithm

In the following, we use the three methods described above to solve the equation

$$\frac{dy}{dx} = -xy \tag{6}$$

which has the exact solution $y(x) = y(0)\exp(-x^2/2)$. Figure 2 shows the difference between the exact and approximate solution for each method, for a step size $\Delta X = .5$, run out to $x = 5$, i.e. 10 steps. The maximum error of the Euler method is somewhat over 0.12, for the midpoint method is about 0.02, and for the Runge-Kutta method is so small that it is not visible on the graph. Direct examination of the results shows the maximum error to be .002. These graphs were produced by the Advanced example script. You should try re-running with smaller Δx (and correspondingly larger number of steps) to see how the results change

But wait – at the larger values of x, the midpoint method actually has larger errors than the Euler method. What is going on? We can get a better idea of the problem by running the solution out to 15 steps, as shown in Fig. 3. we see that the error grows exponentially. This is an example of *numerical instability*. The midpoint method is accurate, but for this particular form of $F(x, y)$ has a spurious instability which grows exponentially, no matter how small the initial amplitude. The instability is spurious in the sense that the original differential equation has no such instability. It is introduced by the numerical approximation.

Instabilities of this sort are dependent on the kind of system one is integrating. In the present case, the instability arises because the decay rate becomes very large when x is large, which requires a small enough Δx to keep the instability in check. In Fig 4, we integrate out to $x = 7.5$ but use a step size of Δx = .1 instead. In this case, the instability is kept under control, at least out to the value of x we have covered.

Also, if you try the system with $F(x, y) = -y$ instead, you will find the midpoint method has no numerical instability.

4 Simple implementation

Under construction. See example script

5 Object-oriented implementation in `Python`

Under construction. See example script

6 Use of `ClimateUtilities` implementation

The object implementation of ODE integration in the `ClimateUtilities` module is called `integrator`. It defines a class that implements Runge-Kutta in-

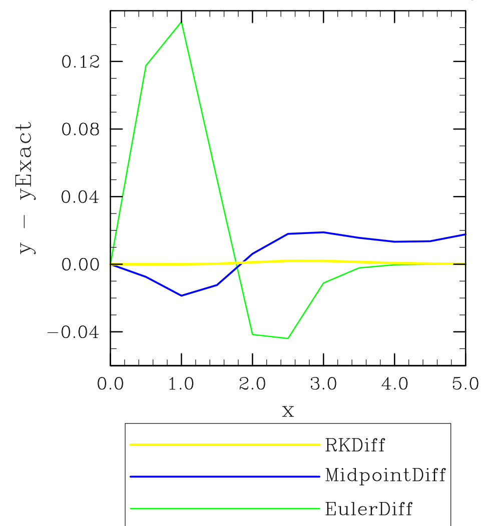

Figure 2:

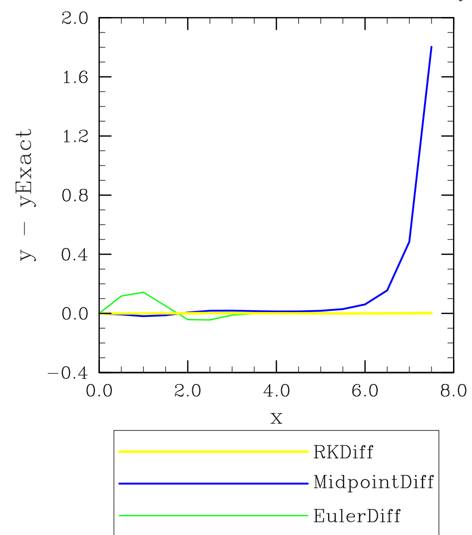

Figure 3:

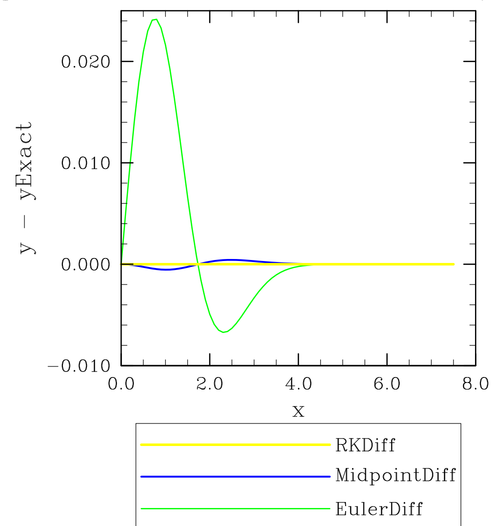

Figure 4:

tegration, and works for 1D problems as well as problems involving multiple dependent variables. It also has features allowing for optionally passing parameters to the slope function. After importing the module, type `help(integrator)` or `help(ClimateUtillities.integrator)`, depending on how you did the import. See the example script for examples of use.

7 Further reading

See *Numerical Recipes*, Cambridge University Press, Chapter 16. The book comes in editions for many computer languages, `Python` unfortunately not yet among them. The algorithms are well described whatever language you choose.