Math 2280–1/2250–4
Numerical Solutions to first order Differential Equations
January 31, 2011

You may wish to download this file from our class homework or lecture page, or by directly opening the URL

from Maple. It contains discussion and Maple commands which will help you answer your homework problems from sections 2.4–2.6.

In this handout we will study numerical methods for approximating solutions to first order differential equations. Later in the course we will see how higher order differential equations can be converted into first order systems of differential equations. It turns out that there is a natural way to generalize what we do now in the context of a single first order differential equations, to systems of first order differential equations. So understanding this material will be an important step in understanding numerical solutions to higher order differential equations and to systems of differential equations.

We will be working through material from sections 2.4–2.6 of the text.

The most basic method of approximating solutions to differential equations is called Euler's method, after the 1700's mathematician who first formulated it. If you want to approximate the solution to the initial value problem

$$\frac{dy}{dx} = f(x, y)$$
$$y(x_0) = y_0,$$

first pick a step size h. Then for x between x_0 and $x_0 + h$, use the constant slope $f(x_0, y_0)$. At x–value $x_1 = x_0 + h$ your y–value will therefore be $y_1 := y_0 + f(x_0, y_0) \, h$. Then for x between x_1 and $x_1 + h$ you use the constant slope $f(x_1, y_1)$, so that at $x_2 := x_1 + h$ your y–value is $y_2 := y_1 + f(x_1, y_1) \, h$. You continue in this manner. It is easy to visualize if you understand the slope field concept we've been talking about; you just use the slope field with finite rather than infinitesimal stepping in the x–variable. You use the value of the slope field at your current point to get a slope which you then use to move to the next point. It is straightforward to have the computer do this sort of tedious computation for you. In Euler's time such computations would have been done by hand!

A good first example to illustrate Euler's method is our favorite DE from the time of Calculus, namely the initial value problem

$$\frac{dy}{dx} = y$$
$$y(0) = 1.$$

We know that $y = e^x$ is the solution. Let's take $h = 0.2$ and try to approximate the solution on the x–interval $[0, 1]$. Since the approximate solution will be piecewise linear, we only need to know the approximations at the discrete x values $x = 0, \ 0.2, 0.4, 0.6, 0.8, 1.0$. I've drawn a picture on the next page with appoximated (x, y) points and the exact the solution graph. Use the empty space to fill in the hand–computations which produce these y values and points. Then compare to the ''do loop'' computation directly below.

approximate
and exact solution graphs
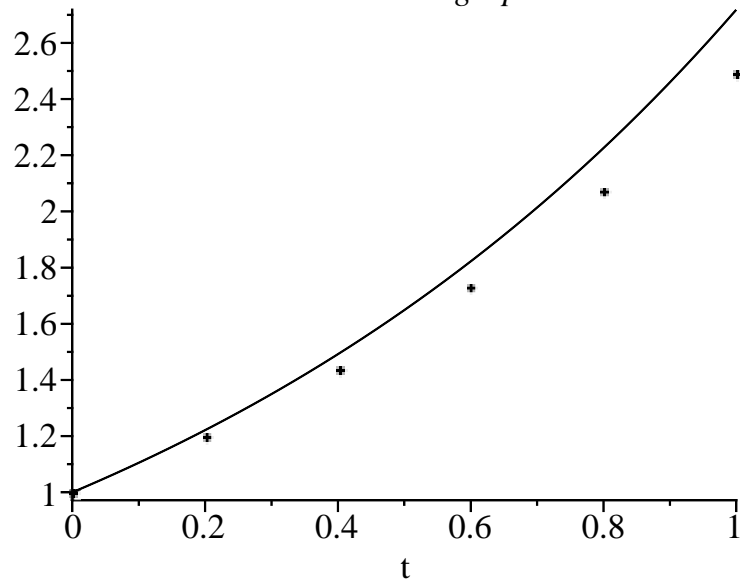
Class exercise 1: Hand work for the approximate solution to

$$\frac{dy}{dx} = y$$

$$y(0) = 1$$

on the interval $[0, 1]$, with $n = 5$ subdivisions, using Euler's method.

Class exercise 2: Why are your approximations too small in this case?

Here is the automated computation:

```
> restart:  #clear any memory from earlier work
> Digits:=6:  #number of significant digits you desire..
       #default is 10, which would clutter up this output.
       #if you wanted, you can ask for a lot more digits too.
> x0:=0.0; xn:=1.0; y0:=1.0; n:=5; h:=(xn-x0)/n;
       # specify initial
       # values, number of steps, step size.
```

$$x0 := 0.$$
$$xn := 1.0$$
$$y0 := 1.0$$
$$n := 5$$
$$h := 0.200000 \tag{1}$$

```
> f:=(x,y)->y;
     #this is the "slope" function f(x,y)
     #in dy/dx = f(x,y).  We want dy/dx = y.
```
$$f := (x, y) \rightarrow y \tag{2}$$

```
> x:=x0; y:=y0;   #initialize x,y for the do loop
```
$$x := 0.$$
$$y := 1.0 \tag{3}$$

```
> for i from 1 to n do
            k:= f(x,y):  #current slope,use: to suppress output
            y:= y + h*k: #new y value via Euler
            x:= x + h:   #updated x-value:
            print(x,y,exp(x));
              #display current values,
              #and compare to exact solution.
     end do:
        #``end do'' ends a do loop
```
$$0.200000, 1.20000, 1.22140$$
$$0.400000, 1.44000, 1.49182$$
$$0.600000, 1.72800, 1.82212$$
$$0.800000, 2.07360, 2.22554$$
$$1.00000, 2.48832, 2.71828 \tag{4}$$

Notice your approximations are all a little too small, in particular your final approximation 2.488... is short of the exact value of $\exp(1) = 2.71828$. Did you figure out why?

The following commands created that graph on page 2. You might want to try understanding and executing them yourself.

```
>  restart:    #clear all memory
>  Digits:=6:
>  with(plots): #load plotting library
   with(LinearAlgebra): #Linear Algebra library
>  f:=(x,y)->y;
       #this is the "slope" function f(x,y)
       #in dy/dx = f(x,y).  We want dy/dx = y.

>  n:=5:h:=1/n:x0:=0:y0:=1: #initialize
>  xval:=Vector(n+1);yval:=Vector(n+1);
       #to collect all our points
>  xval[1]:=x0; yval[1]:=y0;
        #initial values


>    #paste in the previous work, and modify to store
     #all values in an array:
   for i from 1 to n do
              x:=xval[i]:  #current x
              y:=yval[i]:  #current y
              k:= f(x,y):  #current slope
              yval[i+1]:= y + h*k:   #new y value via Euler
              xval[i+1]:= x + h:     #updated x-value:
                 end do:             #ends a do loop
>  approxsol:=pointplot({seq([xval[i],yval[i]], i=1..n+1)}):
>  exactsol:=plot(exp(t),t=0..1,'color'='black'):
        #used t because x was already used above
>  display({approxsol,exactsol},title='approximate
   and exact solution graphs');
```

It should be that as your step size h gets smaller, your approximations to the actual solution get better. This is true if your computer can do exact math (which it can't), but in practice you don't want to make the computer do too many computations because of problems with round–off error and computation time, so for example, choosing $h = .0000001$ would not be practical. But, trying $h = 0.01$ in our previous initial value problem should be instructive.

If we change the n–value to 100 and keep the other data the same we can rerun our experiment:

```
> x0:=0.0; xn:=1.0; y0:=1.0; n:=100; h:=(xn-x0)/n;
  x:=x0; y:=y0;


> for i from 1 to n do
            k:= f(x,y):  #current slope
            y:= y + h*k: #new y value via Euler
            x:= x + h:   #updated x-value:
              if frac(i/10)=0
                then print(x,y,exp(x));
              end if; #use the ``if'' test to decide when to print;
                #the command ``frac'' computes the remainder
                #of a quotient, it will be zero for us if i
                #is a multiple of 10. This way we only print
                #the approximations every 0.1 increments, even
                #though our actual time step is 0.01.
      end do:    #end the do loop
```
$$1.10000, 2.98782, 3.00417$$
$$1.20000, 3.30041, 3.32012$$
$$1.30000, 3.64569, 3.66930$$
$$1.40000, 4.02712, 4.05520$$
$$1.50000, 4.44845, 4.48169$$
$$1.60000, 4.91384, 4.95303$$
$$1.70000, 5.42793, 5.47395$$
$$1.80000, 5.99580, 6.04965$$
$$1.90000, 6.62310, 6.68589$$
$$2.00000, 7.31603, 7.38906$$ **(5)**

So you can see we got closer to the actual value of e, but really, considering how much work Maple did this was not a great result.

Class exercise 3: For this very special initial value problem which has $\exp(x)$ as the solution, set up Euler on the x–interval $[0, 1]$, with n subdivisions, and step size $h = \dfrac{1}{n}$. Write down the resulting Euler estimate for $\exp(1) = e$. What is the limit of this estimate as $n \to \infty$? You learned this special limit in Calculus!

We can make a picture of what we did as follows, using the mouse to cut and paste previous work, and then editing it for the new situation:

```
> restart:
> Digits:=6:with(plots):with(LinearAlgebra):
> f:=(x,y)->y;
  n:=100; x0:=0.0; y0:=1.0;
  xn:=1.0; h:=(xn-x0)/n;
  xval:=Vector(n+1);yval:=Vector(n+1);
     #to collect all our points. Now n=100
  xval[1]:=x0; yval[1]:=y0;
     #initial values
  for i from 1 to n do
          x:=xval[i]:   #current x
          y:=yval[i]:   #current y
          k:= f(x,y):   #current slope
          yval[i+1]:= y + h*k:    #new y value via Euler
          xval[i+1]:= x + h:      #updated x-value:
             od:                  #``od'' ends a do loop
> approxsol2:=pointplot({seq([xval[i],yval[i]], i=1..n+1)}):
  exactsol:=plot(exp(t),t=0..1,'color'='red'):
       #used t because x was already used above
  display({approxsol2,exactsol});
```
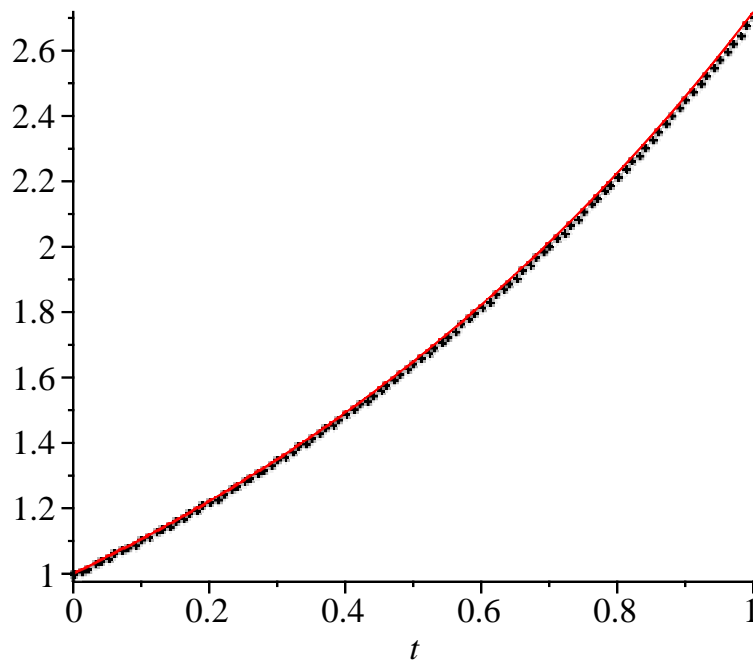
In more complicated problems it is a very serious issue to find relatively efficient ways of approximating solutions. An entire field of mathematics, ''numerical analysis'' deals with such issues for a variety of mathematical problems. The book talks about some improvements to Euler in sections 2.5 and 2.6, in particular it discusses improved Euler, and Runge Kutta. Runge Kutta–type codes are actually used in commerical numerical packages, e.g. in Maple and Matlab.

Let's summarize some highlights from 2.5–2.6.

For any step h the fundamental theorem of calculus asserts that, since $\frac{dy}{dx} = f(x, y)$,

$$y(x + h) = y(x) + \int_x^{x+h} f(t, y(t)) \, dt.$$

The problem with Euler is that we always approximate this integral by $h \cdot f(x, y(x))$, i.e. we use the left–hand endpoint as our approximation of the ''average height''. The improvements to Euler depend on better approximations to that integral. These are subtle, because we don't yet have an approximation for $y(t)$ when t is greater than x, so also not for the integrand. ''Improved Euler'' uses an approximation to the Trapezoid Rule to approximate the integral. Recall, the trapezoid rule for this integral approximation would be $\frac{1}{2}h \cdot (f(x, y(x)) + f(x + h, y(x + h)))$. Since we don't know $y(x + h)$ we approximate it using unimproved Euler, and then feed that into the trapezoid rule. This leads to the improved Euler do loop below. Of course before you use it you must make sure you initialize everything correctly.

```
> restart: Digits:=6: with(plots): with(LinearAlgebra):
> f:=(x,y)->y: #dy/dx=f(x,y)
  x0:=0.: y0:=1.: #initial point
  xn:=1.:  #final x-value
> x:=x0; y:=y0; n:=5; h:=(xn-x0)/n;
> for i from 1 to n do
      k1:=f(x,y):            #left-hand slope
      k2:=f(x+h,y+h*k1):    #approximation to right-hand slope
      k:= (k1+k2)/2:        #approximation to average slope
      y:= y+h*k:             #improved Euler update
      x:= x+h:              #update x
      print(x,y,exp(x));    #exp(x) is exact solution in this case.
  od:
```

$$0.200000, \ 1.22000, \ 1.22140$$
$$0.400000, \ 1.48840, \ 1.49182$$
$$0.600000, \ 1.81585, \ 1.82212$$
$$0.800000, \ 2.21534, \ 2.22554$$
$$1.00000, \ 2.70271, \ 2.71828 \tag{6}$$

Notice you almost did as well with $n = 5$ in improved Euler as you did with $n = 100$ in unimproved Euler.

One can also use Taylor approximation methods to improve upon Euler; by differentiating the equation $\frac{dy}{dx} = f(x, y)$ and using the chain rule on the right hand side one can solve for higher order derivatives of y in terms of the lower order ones, and then use the Taylor approximation for $y(x + h)$ in terms of $y(x)$. See the book for more details of this method, we won't do it here.

In the same vein as "improved Euler" we can use the Simpson approximation for the integral instead of the Trapezoid one, and this leads to the Runge–Kutta method. You may or may not have talked about Simpson's Parabolic Rule for approximating definite integrals in Calculus, it is based on a quadratic approximation to the function f, whereas the Trapezoid rule is based on a first order approximation. In fact, if you fit a parabola $p(x)$ to the three points

$$\left(x_0, f(x_0)\right), \ \left(\frac{(x_0+x_1)}{2}, f\left(\frac{(x_0+x_1)}{2}\right)\right), \ \left(x_1, f(x_1)\right)$$

then

$$\int_{x_0}^{x_1} p(t)\,dt = \frac{(x_1-x_0)}{6}\left(f(x_0) + 4\cdot f\left(\frac{(x_0+x_1)}{2}\right) + f(x_1)\right).$$

That formula is the basis for Simpson's rule, which you can review in your Calculus text or at Wikipedia. (Wikipedia also has a good entry on Runge–Kutta.)

Here is the code for the Runge–Kutta method. The text explains it in section 2.6.

```
> restart: #reinitialize
  Digits:=6: with(plots): with(LinearAlgebra):

> f:=(x,y)->y: #dy/dx=f(x,y)
  x0:=0.: y0:=1.: #initial point
  xn:=1.:  #final x-value
```

We're still using slope function $f(x, y) = y$ but the code below will work for whatever you define this function to be when you do the initialization step.

```
> x := x0 : y := y0 : n := 5 : h := (xn − x0)/n :

> for i from 1 to n do
      k1:=f(x,y):                 #left-hand slope
      k2:=f(x+h/2,y+h*k1/2):   #1st guess at midpoint slope
      k3:=f(x+h/2,y+h*k2/2):   #second guess at midpoint slope
      k4:=f(x+h,y+h*k3):       #guess at right-hand slope
      k:=(k1+2*k2+2*k3+k4)/6:  #Simpson's approximation for the integral
      x:=x+h:                  #x update
      y:=y+h*k:                #y update
      print(x,y,exp(x));       #display current values
    end do:
```

$$0.200000, \ 1.22140, \ 1.22140$$
$$0.400000, \ 1.49182, \ 1.49182$$
$$0.600000, \ 1.82211, \ 1.82212$$
$$0.800000, \ 2.22552, \ 2.22554$$
$$1.00000, \ 2.71825, \ 2.71828$$

(7)

```
>
```

Notice how close Runge–Kutta gets you to the correct value of e, with $n = 5$!

As we know, solutions to non−linear DE's can blow up, and there are other interesting pathologies as well, so if one is doing numerical solutions there is a real need for care and understanding. One such example is in your homework. Another excellent example to look at is Example 4, pages 140−141.