Math 2250-004
Week 4 notes

On Monday Sept 11 we will cover section 2.3, using the notes from last Friday. On Tuesday and Wednesday, possibly creeping into Friday, we will discuss numerical methods for finding solutions to first order differential equations - sections 2.4-2.6. We may have time to begin sections 3.1-3.2 on Friday. If we do, I will post those notes too.

Tuesday Sept 12 - Wednesday Sept 13.
2.4-2.6 numerical methods for solving differential equations

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

http://www.math.utah.edu/~korevaar/2250fall16/week4.mw

from Maple. It contains discussion and Maple commands which will help you with your Maple/Matlab/Mathematica work later this week, in addition to our in-class discussions.

In these notes 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 later on.

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

Euler's Method:

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

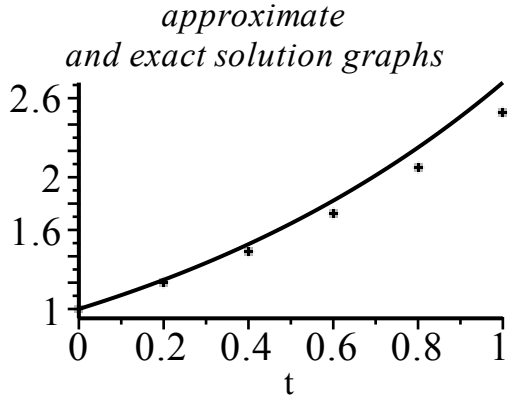
$$\frac{dy}{dx} = f(x, y)$$
$$y(x_0) = y_0,$$

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 $y_1 := y_0 + f(x_0, y_0) \cdot h$. 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 $y_2 := y_1 + f(x_1, y_1) \cdot h$. 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 discrete rather than continuous 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 which is what the application dfield has been doing automatically. (Actually it uses more sophisticated methods to make the steps, related to section 2.6.) 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 already know that $y = e^x$ is the solution so we can compare the numerical techniques to the actual solution function. 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 with appoximated (x, y) points and the exact 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 that follows.



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 and $h = 0.2$, using Euler's method.

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

Here is the automated computation for the numerical computation, and for the graph as well.

```
>
> restart : # clear all memory
> Digits := 6 : #work with 6 significant digits
> with(plots) : # package for plotting
> f := (x, y) → y : #slope function
> n := 5 : h := 1/n : x0 := 0. : y0 := 1. : #initialize
>
> xval[0] := x0 : yval[0] := y0 :
      #initial values. xval and yval will save all approximate points in a list
>  for i from 0 to n do
        print(xval[i], yval[i], e^xval[i]);
        xval[i + 1] := xval[i] + h : #update x
        yval[i + 1] := yval[i] + h·f(xval[i], yval[i]) : #update y
     end do:          #ends a do loop
```

0., 1., 1.

0.200000, 1.20000, 1.22140

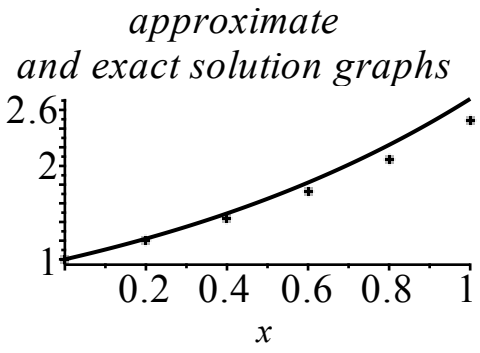
0.400000, 1.44000, 1.49182

0.600000, 1.72800, 1.82212

0.800000, 2.07360, 2.22554

1.00000, 2.48832, 2.71828 **(1)**

```
> #  Create the graph:
> approxsol := pointplot({seq([xval[i], yval[i]], i = 0..n)}) : #(x,y) points from Euler
> exactsol := plot(exp(x), x = 0..1, `color` = `black`) :  #graph of exponential function
> 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:

```
> restart :
> f := (x, y) → y :  # for the DE y'(x)=y
> x0 := 0.0; xn := 1.0; y0 := 1.0; n := 100; h := (xn - x0)/n;
```

$$x0 := 0.$$
$$xn := 1.0$$
$$y0 := 1.0$$
$$n := 100$$
$$h := 0.01000000000 \quad \textbf{(2)}$$

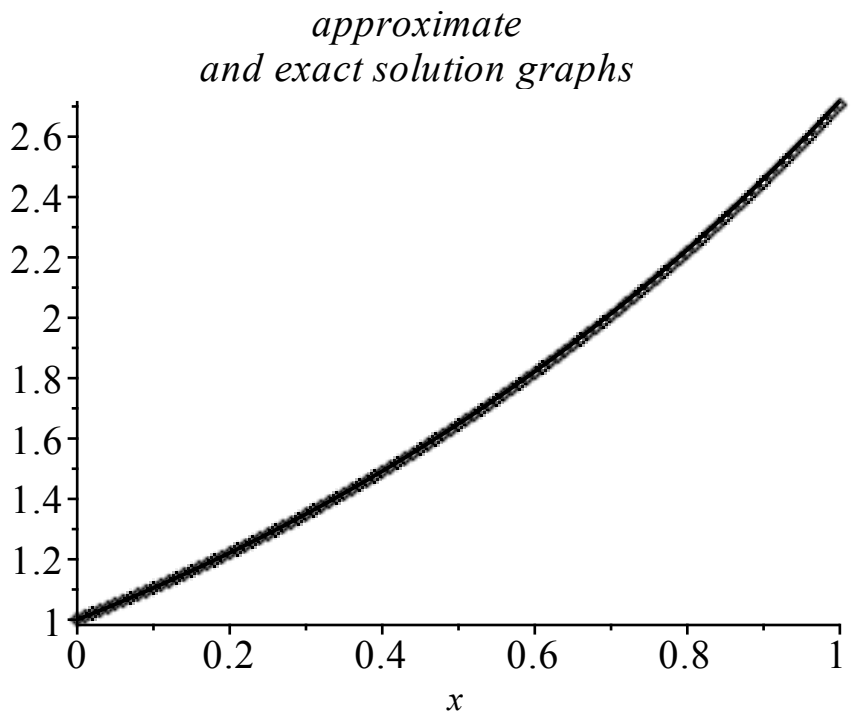
```
>
> xval[0] := x0 : yval[0] := y0 :
      #initial values. xval and yval will save all approximate points in a list
> for i from 0 to n do
      if i mod 10 = 0  # this is asking if "i" is a multiple of 10
                       # i.e. if h is a multiple of 0.1
       then print(xval[i], yval[i], exp(xval[i])) ;
    #current x and y values, and value of exact solution in this case
      end if:
      xval[i + 1] := xval[i] + h :  #update x
      yval[i + 1] := yval[i] + h·f(xval[i], yval[i]) : #update y
        end do:        #ends a do loop
```

0., 1.0, 1.
0.1000000000, 1.104622126, 1.105170918
0.2000000000, 1.220190040, 1.221402758
0.3000000000, 1.347848915, 1.349858808
0.4000000000, 1.488863734, 1.491824698
0.5000000000, 1.644631822, 1.648721271
0.6000000000, 1.816696698, 1.822118800
0.7000000000, 2.006763369, 2.013752707
0.8000000000, 2.216715219, 2.225540928
0.9000000000, 2.448632677, 2.459603111
1.000000000, 2.704813833, 2.718281828 **(3)**

```
>
```

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.

```
> with(plots):
> approxsol := pointplot({seq([xval[i], yval[i]], i = 0..n)}): #(x,y) points from Euler
> exactsol := plot(exp(x), x = 0..1, `color` = `black`): #graph of exponential function
> display({approxsol, exactsol}, title = `approximate
  and exact solution graphs`);
```




```
>
```

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 $h = \frac{1}{n}$. 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!

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, Matlab and dfield.

Let's summarize some highlights from 2.5-2.6.

Suppose we already knew the solution $y(x)$ to the initial value problem

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

If we integrate both sides of the DE from x to $x + h$ and apply the Fundamental Theorem of Calculus, we get

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

The problem with Euler is that we always approximate this integral above by $h \cdot f(x, y(x))$, i.e. we use the left-hand endpoint as our approximation of the ``average height". This causes systematic errors that accumulate as we move from subinterval to subinterval, causing our approximate solution values to diverge away from the actual solution values. 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 $f(t, y(t))$.

``Improved Euler" uses an approximation to the Trapezoid Rule to approximate the $\int_x^{x+h} f(t, y(t))\, dt$.

Recall, the trapezoid rule for this integral approximation would be the width of the interval, times the average of the endpoint values of the integrand,

$$\frac{1}{2} h \cdot (f(x, y(x)) + f(x+h, y(x+h))).$$

We approximate $y(x+h)$ we approximate it using unimproved Euler and our approximations for $y(x)$, and then feed that into the second term of the trapezoid rule. This leads to the improved Euler do loop below. We'll continue to use the exponential growth differential equation from the earlier examples.

```
> 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;
```

$$x := 0.$$

$$y := 1.$$

$$n := 5$$

$$h := 0.200000 \qquad \textbf{(4)}$$

```
>
> xval[0] := x0 : yval[0] := y0 :
      #initial values. xval and yval will save all approximate points in a list
>  for i from 0 to n do
      print(xval[i], yval[i], e^x);
      k1 := f(xval[i], yval[i]) :        #left-hand slope
      k2 := f(xval[i] + h, yval[i] + h * k1) :
    #approximation to right-hand slope for improved Euler
      k := (k1 + k2) / 2 :      #approximation to average slope, for trapezoid-like rule
      xval[i + 1] := xval[i] + h : #update x
      yval[i + 1] := yval[i] + h·k : #update y
      end do:          #ends a do loop
```

$$0., 1., 1.$$

$$0.200000, 1.22000, 1.$$

$$0.400000, 1.48840, 1.$$

$$0.600000, 1.81585, 1.$$

$$0.800000, 2.21534, 1.$$

$$1.00000, 2.70271, 1. \qquad \textbf{(5)}$$

```
>
```

Notice we 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 g, whereas the Trapezoid rule is based on a first order approximation. One of your homework problems from this week reviews where Simpson's rule comes from. In fact, consider two numbers $x_0 < x_1$, with interval width $h = x_1 - x_0$ and interval midpoint $\underline{x} = \frac{1}{2}(x_0 + x_1)$. If you fit a parabola $p(x)$ to the three points

$$\left(x_0, g\left(x_0\right)\right), \quad \left(\underline{x}, g\left(\underline{x}\right)\right), \quad \left(x_1, g\left(x_1\right)\right)$$

then the integral of that parabola over the interval can be computed exactly by the formula

$$\int_{x_0}^{x_1} p(t)\,dt = \frac{h}{6}\left(g\left(x_0\right) + 4 \cdot g\left(\underline{x}\right) + g\left(x_1\right)\right).$$

(You will check this fact in your homework this week!) 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.) Applying the Simpson's rule approximation for our DE we would have

$$y\left(x + h\right) = y(x) + \int_{x}^{x+h} f(t, y(t))\,dt$$

$$\approx y(x) + \frac{h}{6} \cdot \left(f(x, y(x)) + 4 f\left(x + \frac{h}{2}, y\left(x + \frac{h}{2}\right)\right) + f(x + h, y(x + h))\right) \quad .$$

Here's how Runge-Kutta approximates the last two terms in the expression above. The text explains it in section 2.6.

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

```
> 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, depending on the slope function for the DE you're trying to find approximate solutions for.

```
> xval[0] := x0 : yval[0] := y0 : n := 5 : h := (xn - x0)/n :
> for i from 0 to n do
      print(xval[i], yval[i], e^xval[i]);
      x := xval[i] : y := yval[i] :    #x,y temporary placeholders for left endpoint.
      k1 := f(x, y) :                  #left-hand slope
      k2 := f(x + h/2, y + h·k1/2) :   #1st guess at midpoint slope function
      k3 := f(x + h/2, y + h·k2/2) :   #second guess at midpoint slope function
      k4 := f(x + h, y + h·k3) :       #guess at right-hand slope function
      k := (k1 + 2·k2 + 2·k3 + k4)/6 :
    #final guess at average slope function, related to Simpson's Rule
      xval[i + 1] := xval[i] + h :  #update x
      yval[i + 1] := yval[i] + h·k : #update y
    end do:          #ends a do loop
```

0., 1., 1.

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 **(6)**

```
>
```

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, see e.g. p. 140 -141.