Math 2280-001

Wed Feb 1 These are the filled in notes.
2.4-2.6 numerical methods for first order differential equations. We will probably meet in LCB 115 (TBA) to work through these notes, in Maple.

http://www.math.utah.edu/~korevaar/2280spring17/week4.mw

.....

Math Department login for students is as follows:

login name: Suppose your name is Public, John Q. Then your login name is

c-pcjq

(If several UU math students have the same initials as you, then you may actually be signed up as one of c-pcjq1, c-pcjq2, c-pcjq3 or c-pcjq4.)

password: If your UID is u***4986 then your password is

pcjq4986

(unless you've changed it). Even if you had to add a number to your login name, your password will only have the four letters part of your login name.

To open this document from Maple, go to our Math 2280 lecture page, then to the link above. Use the File/Open URL option inside Maple, and copy and paste the URL into the dialog box.

save a copy of this file to your home directory (using whatever name seems appropriate).

.....

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.

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. Consider the initial value problem

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

We make repeated use of the familiar approximation

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x}$$

with fixed "step size" $\Delta x := h$ at our discretion (the smaller the better, probably, if we want to be accurate). We consider approximating the graph of the solution to the IVP. Begin at the initial point (x_0, y_0). Let

$$x_1 = x_0 + h$$

To get the Euler approximation y_1 to the exact value $y(x_1)$ use the rate of change $f(x_0, y_0)$ at your initial point (x_0, y_0), i.e. $\Delta y \approx f(x_0, y_0)\Delta x$, so

$$y_1 = y_0 + f(x_0, y_0)h.$$

In general, if we've approximated (x_j, y_j) we set

$$x_{j+1} = x_j + h$$
$$y_{j+1} = y_j + f(x_j, y_j)h.$$

We could also approximate in the $-x$ direction from the initial point (x_0, y_0), by defining e.g.

$$x_{-1} = x_0 - h$$
$$y_{-1} = y_0 - hf(x_0, y_0)$$

and more generally via

$$x_{-j-1} = x_{-j} - h$$
$$y_{-j-1} = y_{-j} - hf(x_{-j}, y_{-j})$$

...etc.

.

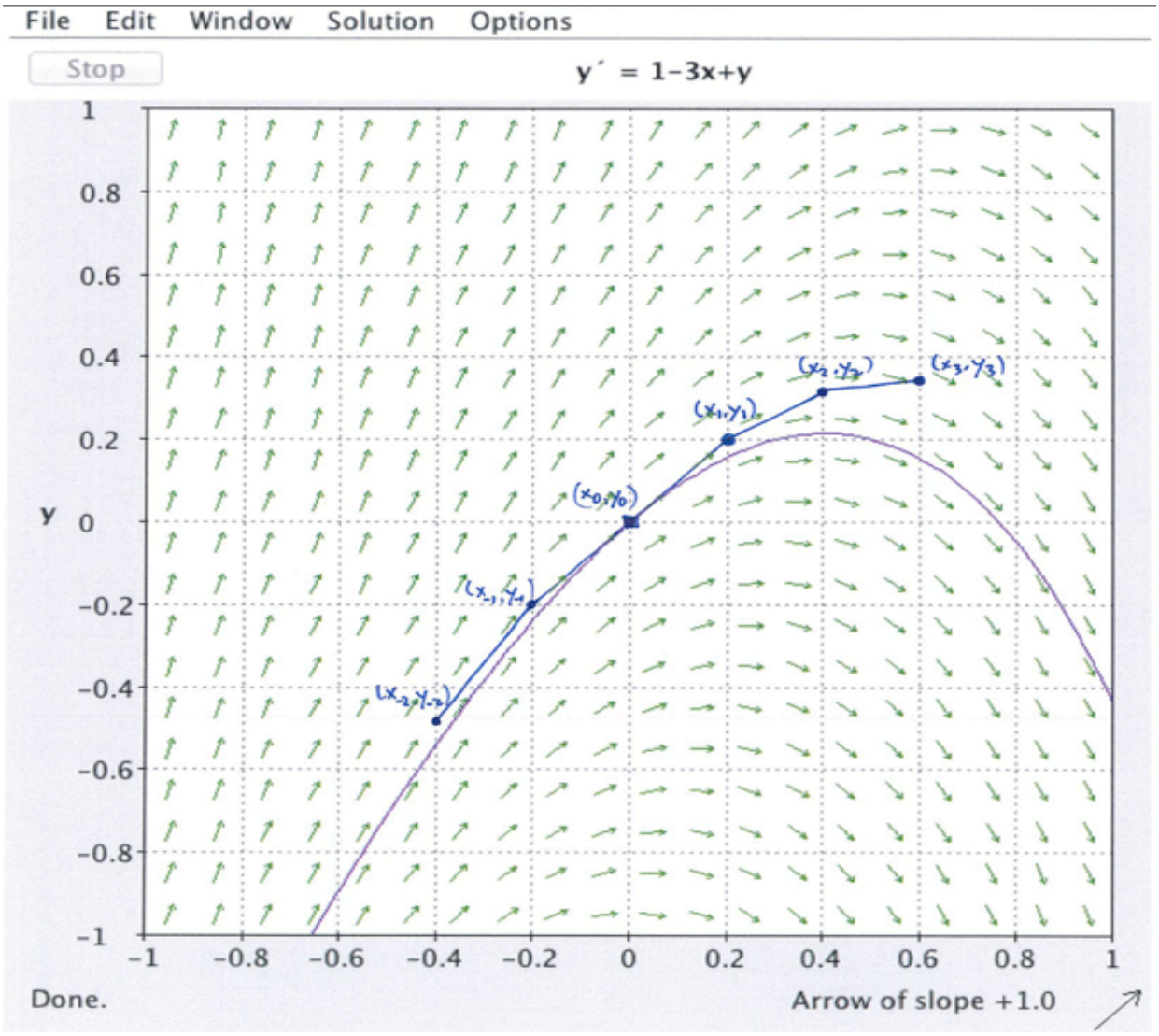
Below is a graphical representation of the Euler method, illustrated for the IVP

$$y'(x) = 1 - 3x + y$$
$$y(0) = 0$$

with step size $h = 0.2$.

Exercise 1 Find the numerical values for several of the labeled points.

$$x0 = 0, y0 = 0, f(x0, y0) = 1$$
$$x1 = .2, y1 = y0 + h \cdot f(x0.y0) = 0 + .2 \cdot 1 = .2, f(.2, .2) = 1 - .6 + .2 = .6$$
$$x2 = .4, y2 = y1 + f(x1, y1) \cdot h = .2 + .6 \cdot 2 = .32$$



As a running example in the remainder of these notes we will use one of our favorite IVP's from the time of Calculus, namely the initial value problem for $y(x)$,

$$y'(x) = y$$
$$y(0) = 1.$$

We know that $y = e^x$ is the solution.

Exercise 2 Work out by hand the approximate solution to the IVP above on the interval $[0, 1]$, with $n = 5$ subdivisions and $h = 0.2$, using Euler's method. This table might help organize the arithmetic. In this differential equation the slope function is $f(x, y) = y$ so is especially easy to compute.

step i	x_i	y_i	$k = f(x_i, y_i) = y_i$	$\Delta x = h$	$\Delta y = h \cdot k$	$x_i + \Delta x$	$y_i + \Delta y$
0	0	1	1	0.2	0.2	0.2	1.2
1	0.2	1.2	1.2	.2	$.2 \cdot 1.2 = .24$	.4	$1.2 + .24 = 1.44$
2	.4	1.2^2	1.2^2	.2	$.2 \cdot 1.2^2$	.6	$1.44 + .288 = 1.728 = 1.2^3$
3						.8	1.2^4
4						1	1.2^5
5	1	$1.2^5 = 2.488$					

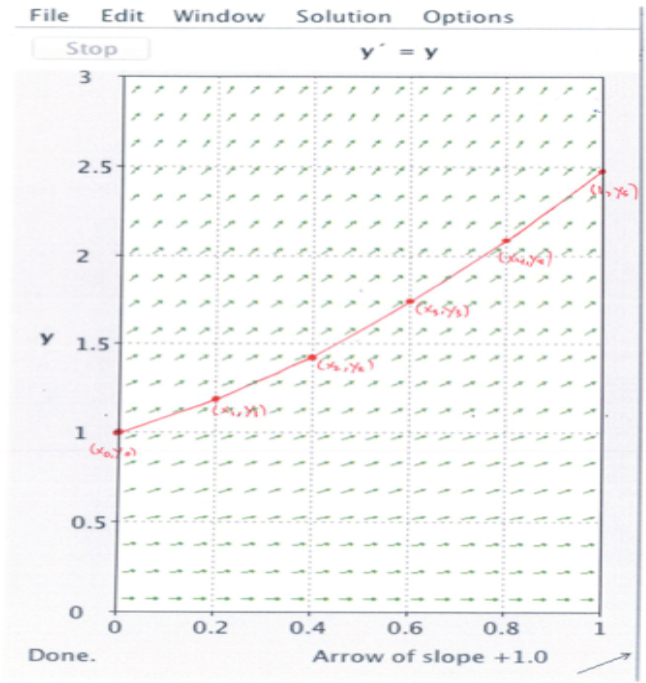
Euler Table

Your work should be consistent with the picture below.

```
> 1.2^5;
```

2.48832 **(1)**

```
>
```


File Edit Window Solution Options
Stop
y' = y
3
2.5
2
1.5
1
0.5
0
y
0
0.2
0.4
0.6
0.8
1
Done.
Arrow of slope +1.0

Here is the automated computation for the previous exercise, and a graph comparing the approximate solution points to the actual solution graph:

Exercise 3 Enter the following commands (by putting your cursor in the command fields and hitting enter). Discuss what each command is doing, and ask for any needed clarification. The first collection of commands is initializing the data. The second set is running the Euler loop. If you forget to initialize the data first, you might run into trouble trying to run the second set. No matter what is written in the document, commands are not executed until the cursor is put into a command field, and the <enter> or <return> key is pressed.

Initialize:

```
> restart: #clear all memory, if you wish
> Digits := 6: #use floating point arithmetic with 6 digits. could use more if desired
> unassign(`x`, `y`); # in case you used the letters elsewhere
> f := (x, y) -> y; # slope field function for the DE y'(x)=y
                    # change for different DE's!!!
```

$$f := (x, y) \to y \qquad (2)$$

```
> x[0] := 0; y[0] := 1; #initial point
  h := 0.2; n := 5;   #step size and number of steps
```

$$x_0 := 0$$

$$y_0 := 1$$

$$h := 0.2$$

$$n := 5 \qquad (3)$$

```
> exactsol := x -> e^x; #exact solution for the IVP y'=y, y(0)=1
                        #change (or omit) for different IVP's!!!
```

$$exactsol := x \to e^x \qquad (4)$$

```
> g := (x, y) -> y^2 + x^3;
```

$$g := (x, y) \to y^2 + x^3 \qquad (5)$$

```
> g(2, 3);
```

$$17 \qquad (6)$$

```
>
```

Euler Loop

```
> for i from 0 to n do #this is an iteration loop, with index "i" running from 0 to n
    print(i, x[i], y[i], exactsol(x[i]));
        #print iteration step, current x,y, values, and exact solution value
    k := f(x[i], y[i]); #current slope function value
    x[i + 1] := x[i] + h;
    y[i + 1] := y[i] + h·k;
  end do: #how to end a for loop in Maple
```

0, 0, 1, 1

1, 0.2, 1.2, 1.22140

2, 0.4, 1.44, 1.49182

3, 0.6, 1.728, 1.82212

4, 0.8, 2.0736, 2.22554

5, 1.0, 2.48832, 2.71828 **(7)**

```
> x[0];
```

0 **(8)**

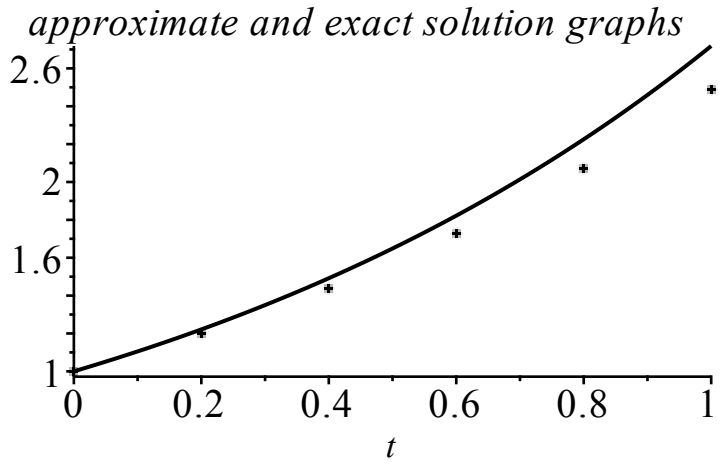
```
> x[5];
```

1.0 **(9)**

```
>
```

Plot results:

```
> with(plots):
  Eulerapprox := pointplot({seq([x[i], y[i]], i = 0..n)}): #approximate soln points
   exactsolgraph := plot(exactsol(t), t = 0..1, `color` = `black`): #exact soln graph
       #used t because x has been defined to be something else
   display({Eulerapprox, exactsolgraph}, title = `approximate and exact solution graphs`);
```




```
>
```

Exercise 4: Why are your approximations too small in this case, compared to the exact solution?

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, copying, pasting, modifying previous work.

Initialize

```
> restart : #clear all memory, if you wish
> unassign( `x`, `y`); # in case you used the letters elsewhere
> Digits := 8 :
> f := (x, y) →y; # slope field function for the DE  y'(x)=y
                  # change for different DE's!!!
```

$$f := (x, y) \to y \tag{10}$$

```
> x[0] := 0; y[0] := 1; #initial point
  h := 0.001; n := 1000;   #step size and number of steps
```

$$x_0 := 0$$

$$y_0 := 1$$

$$h := 0.001$$

$$n := 1000 \tag{11}$$

```
> exactsol := x→e^x; #exact solution for the IVP  y'=y, y(0)=1
                     #change (or omit) for different IVP's!!!
```

$$exactsol := x \to e^{x} \tag{12}$$

```
>
```

Euler Loop (modified using an "if then" conditional clause to only print every 10 steps)

```
> for i from 0 to n do  #this is an iteration loop, with index "i" running from 0 to n
      if frac( i/100 ) = 0
          then print(i, x[i], y[i], exactsol(x[i]) );
      end if: #only print every tenth time
    k := f(x[i], y[i]); #current slope function value
    x[i + 1] := x[i] + h;
    y[i + 1] := y[i] + h·k;
  end do: #how to end a for loop in Maple
```

0, 0, 1, 1

100, 0.100, 1.1051155, 1.1051709

200, 0.200, 1.2212801, 1.2214028

300, 0.300, 1.3496561, 1.3498588

400, 0.400, 1.4915260, 1.4918247

500, 0.500, 1.6483085, 1.6487213

600, 0.600, 1.8215721, 1.8221188

700, 0.700, 2.0130479, 2.0137527

800, 0.800, 2.2246510, 2.2255409

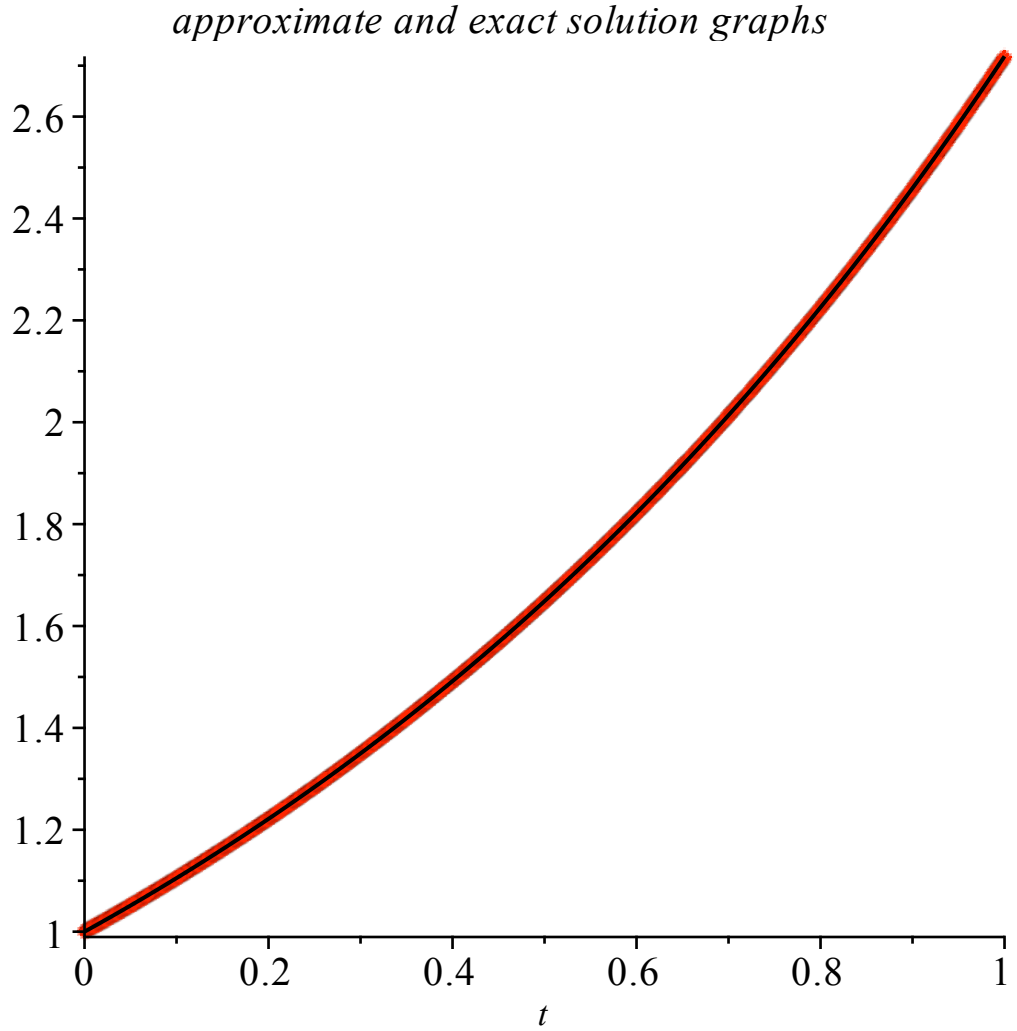
900, 0.900, 2.4584969, 2.4596031

1000, 1.000, 2.7169231, 2.7182818 **(13)**

>

plot results:

```
> with(plots):
  Eulerapprox := pointplot({seq([x[i], y[i]], i = 0..n)}, color = red):
   exactsolgraph := plot(exactsol(t), t = 0..1, `color` = `black`):
      #used t because x was already used has been defined to be something else
   display({Eulerapprox, exactsolgraph}, title = `approximate and exact solution graphs`);
```



>

Exercise 5: For this very special initial value problem

$$y'(x) = y$$
$$y(0) = 1$$

which has $y(x) = e^x$ 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 differential equations 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. Our text explores 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.

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

$$y'(x) = f(x, y)$$
$$y(x_0) = y_0.$$

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

$$y(x + h) - y(x) = \int_x^{x+h} y'(t)\, dt, \text{ i.e.}$$

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

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

One problem with Euler is that we approximate this integral above by $h \cdot f(x, y(x))$, i.e. we use the value at the left-hand endpoint as our approximation of the integrand, on the entire interval from x to $x + h$. This causes errors that are larger than they need to be, and these errors accumulate as we move from subinterval to subinterval and as our approximate solution diverges from the actual solution. The improvements to Euler depend on better approximations to the 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 $f(t, y(t))$ on the interval $[x, x + h]$.

Improved Euler uses an approximation to the Trapezoid Rule to compute the integral in the formula

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

Recall, the trapezoid rule to approximate the integral

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

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 its value with unimproved Euler, and substitute into the formula above. This leads to the improved Euler "pseudocode" for how to increment approximate solutions.

Improved Euler pseudocode:

$k_1 = f(x_j, y_j)$ #current slope

$k_2 = f(x_j + h, y_j + h\, k_1)$ #use unimproved Euler guess for $y(x_j + h)$ to estimate slope function when $x = x_j + h$

$k = \frac{1}{2}(k_1 + k_2)$ #average of two slopes

$x_{j+1} = x_j + h$ #increment x

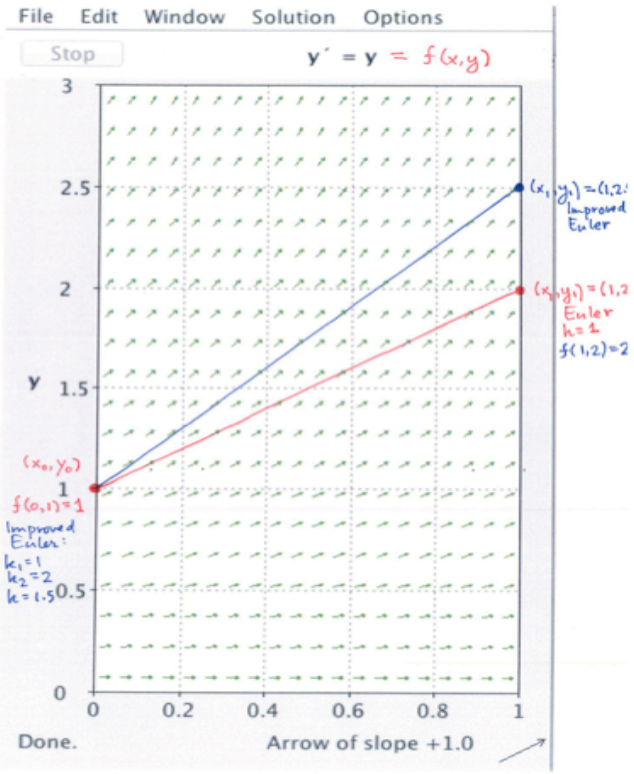
$y_{j+1} = y_j + h\, k$ #increment y

Exercise 6 Contrast Euler and Improved Euler for our IVP

$$y'(x) = y = f(x, y)$$
$$y(0) = 1$$

to estimate $y(1) \approx e$ with one step of size $h = 1$. Your computations should mirror the picture below. Note that with improved Euler you actually get a better estimate for e with ONE step of size 1 than you did with 5 steps of size $h = 0.2$ using unimproved Euler. (It's still not a very good estimate.)

$$x0 = 0, y0 = 1$$
$$k1 = f(0, 1) = 1$$
$$k2 = f(1, 1 + 1 \cdot 1) = f(1, 2) = 2$$
$$k = \frac{(1+2)}{2} = 1.5$$
$$x1 = 1, y1 = 1 + 1 \cdot k = 2.5$$


File Edit Window Solution Options
Stop
y' = y = f(x,y)
3
2.5
2
1.5
1
0.5
0
y
0
0.2
0.4
0.6
0.8
1
(x0, y0)
f(0,1)=1
Improved Euler:
k1=1
k2=2
k=1.5
(x1, y1)=(1,2
Improved Euler
(x1, y1)=(1,2
Euler
h=1
f(1,2)=2
Done.
Arrow of slope +1.0

In the same vein as ``improved Euler" we can use the Simpson approximation for the integral for Δy instead of the Trapezoid rule, 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 integrand g, whereas the Trapezoid rule is based on "linear" approximation.

Simpson's Rule:

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

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

then

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

(You will check this fact in your homework this week!) This formula is the basis for Simpson's rule, which you can also 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, and if we already knew the solution function $y(x)$, we would have

$$y(x+h) - y(x) = \int_x^{x+h} y'(t)\, \mathrm{d}t \text{, i.e.}$$

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

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

However, we have the same issue as in Trapezoid - that we've only approximated up to $y(x)$ so far. Here's the pseudo-code for how Runge-Kutta takes care of this. (See also section 2.6 to the text.)

Runge-Kutta pseudocode:

$k_1 = f\left(x_j, y_j\right)$ # left endpoint slope

$k_2 = f\left(x_j + \frac{h}{2}, y_j + \frac{h}{2} k_1\right)$ # first midpoint slope estimate, using k_1 to increment y_j

$k_3 = f\left(x_j + \frac{h}{2}, y_j + \frac{h}{2} k_2\right)$ # second midpoint slope estimate using k_2 to increment y_j

$k_4 = f\left(x_j + h, y_j + h\, k_3\right)$ # right hand slope estimate, using k_3 to increment y_j

$k = \frac{1}{6}\left(k_1 + 2\, k_2 + 2\, k_3 + k_4\right)$ #weighted average of all four slope estimates, consistent with Simpson's rule

$x_{j+1} = x_j + h$ # increment x

$y_{j+1} = y_j + h\, k$ # increment y

Exercise 7 Contrast Euler and Improved Euler to Runge-Kutta for our IVP

$$y'(x) = y = f(x, y)$$
$$y(0) = 1$$

to estimate $y(1) \approx e$ with one step of size $h = 1$. Your computations should mirror the picture below. Note that with Runge Kutta you actually get a better estimate for e with ONE step of size $h = 1$ than you did with 100 steps of size $h = 0.01$ using unimproved Euler!

$$x0 = 0, y0 = 1, k1 = 1,$$
$$k2 = f(.5, 1.5) = 1.5, k3 = f(.5, 1 + .5 \cdot 1.5) = 1.75$$
$$k4 = f(1, 1 + 1.75) = 2.75$$
$$k = \frac{1}{6}(1 + 3 + 3.5 + 2.75)$$
$$x1 = 1, y1 = 1 + 1 \cdot 1.70833 = 2.70833$$

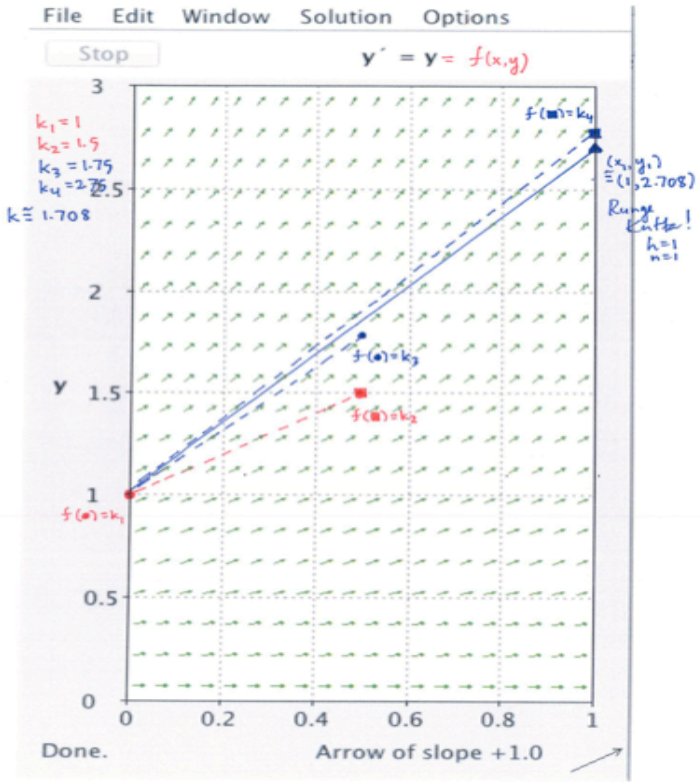
```
> 1/6*(1+3+3.5+2.75);
```

$$1.7083333 \quad \textbf{(14)}$$

```
> evalf(e);
```

$$2.7182818 \quad \textbf{(15)}$$

```
>
```



Numerical experiments and homework

You can do the section 2.4-2.6 homework this week using the code snippets below. I'll illustrate by doing the "famous numbers" problem of estimating e.

Estimate e *by estimating* $y(1)$ for the solution to the IVP

$$y'(x) = y$$
$$y(0) = 1.$$

Apply Runge-Kutta with n = 10, 20, 40... subintervals, successively doubling the number of subintervals until you obtain the target number below - rounded to 9 decimal digits - twice in succession.

```
> evalf(e);
```

$$2.718281828459045 \quad \textbf{(16)}$$

Initialize

```
> restart: # clear any memory from earlier work
> Digits := 20: # we need lots of digits for "famous numbers"
>
> unassign(`x`, `y`); # in case you used the letters elsewhere, and came back to this piece of code
> f := (x, y) -> y; # slope field function for our DE i.e. for y'(x)=f(x,y)
```

$$f := (x, y) \to y \quad \textbf{(17)}$$

```
> x[0] := 0; y[0] := 1; #initial point
  h := 0.01; n := 100;   #step size and number of steps - first attempt for "famous number e"
```

$$x_0 := 0$$
$$y_0 := 1$$
$$h := 0.01$$
$$n := 100 \quad \textbf{(18)}$$

Runge Kutta loop. WARNING: ONLY RUN THIS CODE AFTER INITIALIZING

```
> for i from 0 to n do #this is an iteration loop, with index "i" running from 0 to n
    if frac(i/10) = 0
      then print(i, x[i], y[i], exactsol(x[i]));
    end if: #only print every tenth time

    k1 := f(x[i], y[i]); #current slope function value
    k2 := f(x[i] + h/2, y[i] + h/2 * k1);
     # first estimate for slope at right endpoint of midpoint of subinterval
    k3 := f(x[i] + h/2, y[i] + h/2 * k2); #second estimate for midpoint slope
    k4 := f(x[i] + h, y[i] + h*k3); #estimate for right-hand slope
```

```
    k := (k1 + 2·k2 + 2·k3 + k4)/6; # Runge Kutta estimate for rate of change of y
    x[i + 1] := x[i] + h;
    y[i + 1] := y[i] + h·k;
end do: #how to end a for loop in Maple
```

0, 0, 1, *exactsol*(0)

10, 0.10, 1.1051709180665142871, *exactsol*(0.10)

20, 0.20, 1.2214027581399820355, *exactsol*(0.20)

30, 0.30, 1.3498588075425366526, *exactsol*(0.30)

40, 0.40, 1.4918246975919554529, *exactsol*(0.40)

50, 0.50, 1.6487212706320014536, *exactsol*(0.50)

60, 0.60, 1.8221188003001590068, *exactsol*(0.60)

70, 0.70, 2.0137527073539823382, *exactsol*(0.70)

80, 0.80, 2.2255409283453293374, *exactsol*(0.80)

90, 0.90, 2.4596031109740101131, *exactsol*(0.90)

100, 1.00, 2.7182818282344013785, *exactsol*(1.00) **(19)**

```
> evalf(e);
```

2.71828182845905 **(20)**

For other problems you may wish to use or modify the following Euler and improved Euler loops.

Euler loop: WARNING: ONLY RUN THIS CODE AFTER INITIALIZING

```
> for i from 0 to n do  #this is an iteration loop, with index "i" running from 0 to n
    print(i, x[i], y[i]);
        #print iteration step, current x,y, values, and exact solution value
    k := f(x[i], y[i]);  #current slope function value
    x[i + 1] := x[i] + h;
    y[i + 1] := y[i] + h·k;
  end do:  #how to end a for loop in Maple
>
```

Improved Euler loop: WARNING: ONLY RUN THIS CODE AFTER INITIALIZING

```
> for i from 0 to n do  #this is an iteration loop, with index "i" running from 0 to n
    print(i, x[i], y[i]);  #print iteration step, current x,y, values, and exact solution value
    k1 := f(x[i], y[i]);  #current slope function value
    k2 := f(x[i] + h, y[i] + h·k1 ); #estimate for slope at right endpoint of subinterval
    k := (k1 + k2)/2;  # improved Euler estimate
    x[i + 1] := x[i] + h;
    y[i + 1] := y[i] + h·k;
  end do:  #how to end a do loop in Maple
>
```