Wed Jan 30

2.5-2.6 Improved Euler and Runge Kutta.

Announcements: 2.4-2.6 numerical approx. sol'ns to DE's is an important part of "numerical analysis" applied math (& related fields)

Quiz is on 2.3 (linear drag)

My goal: gentle introduction also to Matlab

I've posted files to CANVAS

I'll try to reserve LCB 115 for Friday 2-3. to help you get started.

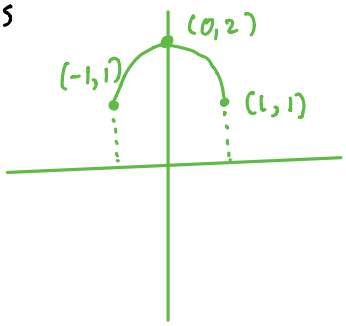
Warm-up Exercise:

Find the quadratic function $q(x) = ax^2 + bx + c$

a) So that the points

$(-1, 1), (0,2), (1,1)$

lie on the graph of q. In other words



$$\begin{cases} q(-1) = 1 = a - b + c \\ q(0) = 2 = c \\ q(1) = 1 = a + b + c \end{cases}$$

$a = -1$, $b = 0$, $c = 2$

b) For this $q(x)$, compute $\int_{-1}^{1} q(x)\,dx$

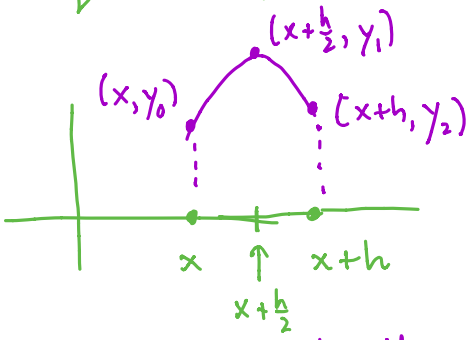
$q(x) = -x^2 + 2$

$q(x) = 2 - x^2$

$$\int_{-1}^{1} q(x)\,dx = \frac{10}{3}$$

$\int_{-1}^{1} q(x)\,dx$ ↑ $2 - x^2$

Fact:



$q(x)$ interpolates those 3 points

$$\int_x^{x+h} q(t)\,dt = \frac{h}{6}\left[y_0 + 4y_1 + y_2\right] = \underbrace{h}_{\text{width interval}}\left[\frac{1}{6}\left(y_0 + 4y_1 + y_2\right)\right]$$

See HW w4.3

in warmup.

b): ans $2\left[\frac{1}{6}(1 + 8 + 1)\right] = \frac{10}{3}$

Runge Kutta

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 parabolic approximation to the integrand , whereas the Trapezoid rule is based on an approximationg of the graph with trapezoids, on small subintervals.

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

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

then

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

(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

$y' = f(x,y)$ $\quad \int_x^{x+h} y'(t) = \int_x^{x+h} f(t, y(t))\,dt$

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

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

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

Runge-Kutta pseudocode:

$k_1 = f(x_j, y_j)$ # 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

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

$k = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ #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 1 Contrast Euler and Improved Euler to Runge-Kutta for our IVP

$$\left.\begin{aligned} y'(x) &= y \\ y(0) &= 1 \end{aligned}\right]$$ Soln $y = e^x$

to estimate $y(1) \approx e$ with one step of size $h = 1$. Your computations should mirror the picture below. Note how amazingly accurate your prediction is, considering you only did a single step. It's actually more accurate than regular Euler with 100 steps of size 0.01.

$x_0 = 0$
$y_0 = 1$
$h = 1$ one step to estimate $e^1 = 2.718\ldots$
(i.e. $x_1 = 1$)

Runge-Kutta pseudocode:

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

$k_1 = f(x_j, y_j)$ # 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

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

$k = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ #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

Euler $y_1 = 2$

Improved Euler $y_1 = 2.5$

$x_0 = 0$
$y_0 = 1$ $h = 1$

$k_1 = f(0,1) = 1$

$k_2 = f(0+.5, \underbrace{1+.5(1)}_{1.5}) = 1.5$

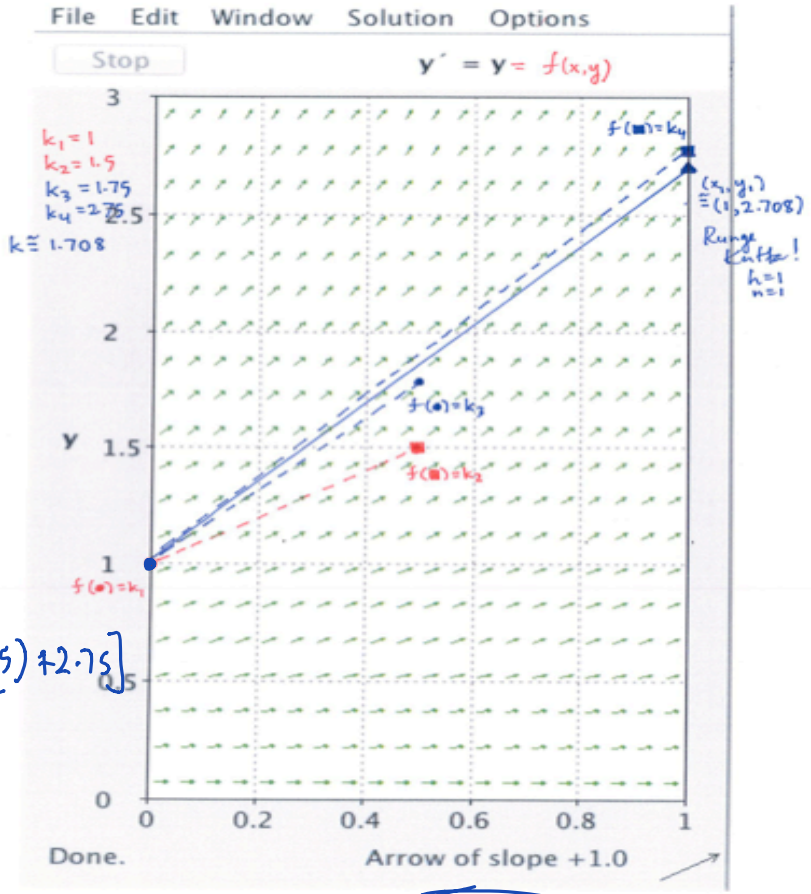
$k_3 = f(.5, \underbrace{1+(.5)(1.5)}_{1.75}) = 1.75$

$k_4 = f(1, 1+1\cdot(1.75)) = 2.75$

$k = \frac{1}{6}\left[1 + \underbrace{2(1.5)}_{3} + \underbrace{2(1.75)}_{3.5} + 2.75\right]$

7.5
2.75
10.25

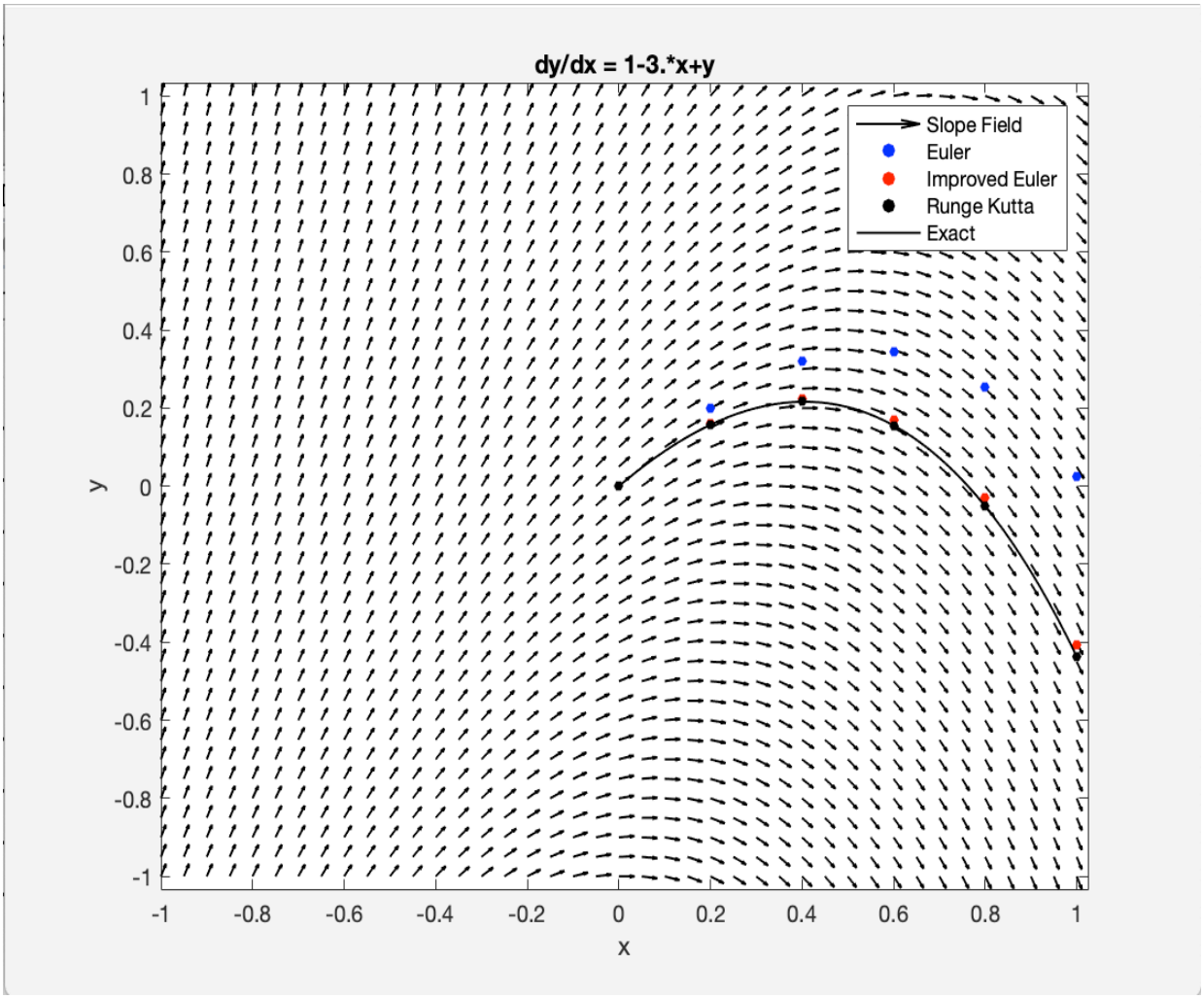
$6 \overline{)\,10.25}$ 1.71
6
4.25
4.2
.05



$y_1 = y_0 + hk = 1 + 1(1.71) = 2.71$

e actually 2.718281828...

Using the Runge-Kutta function on our running example:



```
function [x,y] = Runge_Kutta(x0,y0,b,n,f)
% Runge Kutta method with n subintervals to solve the initial value
% problem dy/dx=f(x,y) with
% initial conditions y(x0)=y0 on the interval [x0,b]. The function
% f(x,y) must be entered as a string.

% For example, if you wish to solve dy/dx=y*cos(x) with y(0)=1 on the
% interval [0,15] with 50 intervals you would enter
% Runge_Kutta(0,1,15,50,'y.*cos(x)')

f = str2func(['@(x,y) ',f]); % converts the string input into a function handle

h=(b-x0)/n;
x=zeros(n+1,1); y= zeros(n+1,1); % Pre-allocoting vectors for speed
x(1)=x0; y(1)=y0;
for i=1:n
    x(i+1)=x0+i*h;
    k1=f(x(i),y(i));
    k2=f(x(i)+h/2,y(i)+h/2*k1);
    k3=f(x(i)+h/2,y(i)+h/2*k2);
    k4=f(x(i+1),y(i)+h*k3);
    y(i+1)=y(i)+h*(k1+2*k2+2*k3+k4)/6;
end
end
```

Numerical experiments (In Maple, which has an easier time than Matlab in adjusting significant digits.)

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 **(5)**

It worked with $n = 40$:

Initialize

```
> restart : # clear any memory from earlier work
> Digits := 15 : # 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)
> x[0] := 0 : y[0] := 1 : #initial point
  h := 0.025 : n := 40 : #step size and number of steps - first attempt for "famous number e"
```

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] ); #print iteration step, current x,y, values, and exact solution value
    end if:
    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

10, 0.250, 1.28402541566434

20, 0.500, 1.64872126807197

30, 0.750, 2.11700001155073

40, 1.000, 2.71828181979283 **(6)**

Matlab

```
%famous numbers

clc, clear, close all

format long
   %display 15 digits
f='y';
   % because we are solving y'(x)=f(x,y) for f(x,y)=y,
   % and estimating y(1), since the solution to y'=y
   % with y(0)=1 is exp(x) and exp(1)=e.
x0 = 0;
y0 = 1;
b=1;
n = 5;

 % Use Runge Kutta method to solve:
          [xR yR]=Runge_Kutta(x0,y0,b,n,f);
 % compare our estimate to actual value of e, as a column vector.
 ans1=[n;yR(n+1);exp(1)]
     % everything in Matlab is a matrix.  Here ans1 is a 3 by 1 matrix,
     % because I used semicolons between the quantities.  If I'd have used
     % commas Matlab would've made a 1 by 3 matrix.
```

……

Results from command window:

```
>> famous_numbers

ans1 =

  10.000000000000000
   2.718279744135166
   2.718281828459046

>> famous_numbers

ans1 =

  20.000000000000000
   2.718281692656335
   2.718281828459046

>> famous_numbers

ans1 =

  40.000000000000000
   2.718281819792855
   2.718281828459046
```