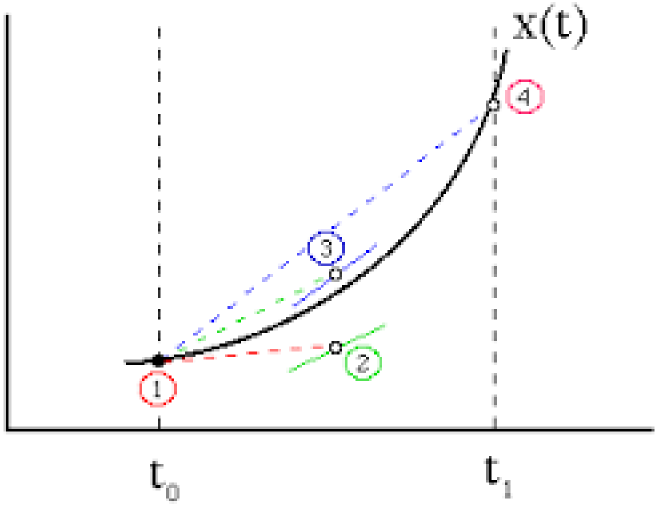
RK:
Runge-Kutta ODE Solvers

MATH1902: Numerical Solution of Differential Equations

http://people.sc.fsu.edu/∼jburkardt/classes/math1902_2020/rk/rk.pdf



A fourth order Runge Kutta step involves several initial test steps.

ODE's by a Runge-Kutta method

If the Euler method requires too many steps, we can select a more accurate solver from the Runge-Kutta family.

1 How accurate is the Euler method?

We are interested in approximately solving an ordinary differential equation with an initial condition:

$$y' = f(t,y) \quad \textit{(a given derivative function we will also call “dydt()”)}$$
$$y(t_0) = y_0 \quad \textit{(a given initial value)}$$

and so far we have used the forward Euler method. Given the pair of values (t_0, y_0) and a stepsize which we call dt or h, the method produces the approximation (t_1, y_1) for the next time, where

$$t_1 = t_0 + dt;$$
$$y_1 = y_0 + dt * dydt(t_0, y_0);$$

Now where did the Euler estimate come from? Let's suppose the exact solution $y(t)$ of the ODE is "reasonably smooth" - in this case, assume it has at least two continuous derivatives over the time interval we are interested in. Then by Taylor's theorem we have

$$y(t_1) = y(t_0) + (t_1 - t_0)\, y'(t_0) + \frac{1}{2}\,(t_1 - t_0)^2\, y''(\xi)$$

for some point $t_0 \leq \xi \leq t_1$.

If we compare the Euler method and the Taylor result, we can evaluate the absolute value of the error in our first Euler step, known as the "local error":

$$\text{Euler local error} = |y(t_1) - y_1| = \frac{1}{2}\, dt^2\, |y"(\xi)|$$

This tells us that if we cut the stepsize in half, the error bound on our first Euler step would be multiplied by a fourth. This suggests why a smaller step size helps. For the Euler method, we say the the local error behaves like dt^2, which we can symbolize by

$$\text{Euler local error} \propto dt^2$$

or we say this local error is $O(dt^2)$, pronounced "order of dt squared".

However, we are interested in approximating the solution from t_0 all the way to a fixed final time. say T, and then estimating the "global error", that is, an upper bound on the absolute value of the difference between our list of approximate values and the exact solution at those times. If we know the local error behavior, we can actually make a statement about the global error.

We can see how this might be true in the case of the Euler method. If we cut the stepsize in half, then we double the number of steps we have to take to reach T. Roughly speaking, we take twice as many steps, each having one quarter the local error, which turns out to give about half the global error at the final time.

Assuming the exact solution is twice continuously differentiable over the interval $[t_0, T]$, then the local error for the forward Euler method is $O(dt^2)$ and the global error is $O(dt)$.

In other words, cutting the stepsize in half will divide the very first error by 4, but overall, this will only tend to reduce the error everywhere by half. Unfortunately, if we have a difficult ODE to solve, and our first attempt at solving it with the Euler method gives us a very large global error, then we may have to reduce the stepsize many times before we see a reasonable approximation. We saw this problem already in the predator/prey ODE.

Thus, if we encounter an ODE for which the Euler method does not seem to be giving us a good approximation, and we suspect that we would need a very small time step to proceed, we should be interested in using other methods which have a smaller local error behavior, because they should get us good results faster.

As it turns out, the Euler method can be regarded as the first member of a family known as Runge-Kutta methods. When we think of it this way, the Euler method can be renamed as *rk1*, where the "1" indicates that it produces a global error characterized as $O(dt^1)$. We will now look at the next Runge-Kutta method, known as *rk2*, for which the global error is $O(dt^2)$. We expect this method to do better on our predator-prey problem.

2 *rk2.m*, an ODE solver with global error $O(dt^2)$

The Euler method worked by evaluating the ODE right hand side at the starting point, and then behaving as this slope was a good estimate of the behavior of the curve over the interval $[t, t + dt]$. If we want an improved solution at $t + dt$, we need a better estimate for the slope over the interval. The Runge-Kutta method *rk2* does this by trying to average the slopes at the two endpoints.

In order to evaluate the slope at the far endpoint, we need a value for y there. We don't have such a value, but we can use the Euler method to estimate it. So in other words, the *rk2* method starts out by using the Euler method to estimate the next solution. It uses the estimated next solution to evaluate the derivative at the far end. Then it averages the near and far derivatives to get a slope estimate over the interval. And it uses that improved estimate to take the step.

Here is a sketch of the process. We assume we have computed a solution estimate (t_n, y_n), which we think of as the "near" data. We want to make an estimate (t_{n+1}, y_{n+1}) at the "far" end of our current interval:

$$\begin{aligned}
f_{near} &= y'(t_n, y_n) && \textit{Slope at near end} \\
y_{far} &= y_n + dt\, y'(t_n, y_n) && \textit{Estimate far solution using Euler step} \\
f_{far} &= y'(t_{n+1}, y_{far}) && \textit{Estimate slope at far end} \\
y_{n+1} &= y_n + dt\,(f_{near} + f_{far})/2 && \textit{Forward step using improved slope estimate}
\end{aligned}$$

Here is a MATLAB version of this algorithm:

```
function [ t, y ] = rk2 ( dydt, tspan, y0, n )

  dt = ( tspan(2) - tspan(1) ) / n;

  t(1,1) = tspan(1);
  y(1,:) = y0(:);

  for i = 1 : n
    f1 = dydt ( t(i,1),      y(i,:)           );
    f2 = dydt ( t(i,1) + dt, y(i,:) + dt * f1' );
    t(i+1,1) = t(i,1) + dt;
    y(i+1,:) = y(i,:) + dt * ( f1' + f2' ) / 2.0;
  end

  return
end
```

Listing 1: MATLAB code for rk2 method.

Try to create a copy of *rk2.m* **yourself.** Make sure you think about what is going on in each line. In particular, notice that the solution vector `y()` is written as a 2-dimensional vector. What is stored along a row, and what is stored along a column?

Because the derivative vectors `f1` and `f2` are column vectors, we have to transpose with an apostrophe sign when we want to form expressions with the row vector information in `y`. MATLAB makes a distinction between row and column vectors, and the MATLAB ODE interface expects column vectors for derivatives and row vectors for individual solutions. Don't worry too much about this for now, but you should start becoming aware of how MATLAB vectors work, and be able to identify the difference between row and column vectors.

3 *predator_rk2.m* uses rk2 to solve the predator ODE

Because the *rk2.m* code has the same input and output format as *euler.m*, we can rerun the predator/prey ODE with this new solver. We can make a copy of *predator_euler.m* and call it *predator_rk2.m*, and replace the word *euler* by *rk2*. In particular, our call to the ODE solver becomes:

```
[ t, y ] = rk2 ( dydt, tspan, y0, n );
```

Listing 2: Calling the rk2 ODE solver.

For the *euler()* solver, we had to use as many as $n = 100,000$ steps to get a reasonable solution to the predator prey problem. Because the global error in the *rk2()* solver behaves quadratically, we could hope to get equivalent accuracy with somewhere between 100 and 1,000 steps. *Can you explain why a method of order 2 might be as accurate as a method of order 1, while using roughly the square root of the number of steps?*

Investigate this claim! Use the *rk2* solver on the predator/prey problem with steps $n = 100, 200, 300, ...$ until you get a solution that you believe is accurate.

To choose a satisfactory value of n, consider the evidence of the three types of plots:

1. the plots of y(t) and y'(t) should each have repeated peaks of almost the same height;
2. the phase plot of y(t) versus y'(t) should look like a smooth closed curve;
3. the convervation plot should show very litle change.

The conservation plot should give you the best report on whether the solution is accurate.

You should be able to conclude that *rk2()* got you a good answer with much less work than the Euler method aka *rk1()*.

4 *rk4.m*, an ODE solver with global error $O(dt^4)$

If we are interested in even higher accuracy, there is a sequence of Runge-Kutta ODE solvers of increasing order. One of the popular methods is *rk4*, whose global error tends to behave like $O(dt^4)$. We will use this method in a few examples; the user input/output interface is the same as we have seen for *euler/rk1* and *rk2*. On the other hand, the actual MATLAB code is complicated and more difficult to explain, so we will simply display it, and remark that it uses four samples of the right hand side function in order to make a very accurate approximation of the next solution.

```
function [ t, y ] = rk4 ( dydt, tspan, y0, n )

  m = length ( y0 );
  t = zeros ( n + 1, 1 );
  y = zeros ( n + 1, m );

  t0 = tspan(1);
  tstop = tspan(2);
  dt = ( tstop - t0 ) / n;

  t(1,1) = t0;
  y(1,:) = y0(:);

  for i = 1 : n

    f1 = dydt ( t(i,1),            y(i,:) );
    f2 = dydt ( t(i,1) + dt / 2.0, y(i,:) + dt * f1' / 2.0 );
    f3 = dydt ( t(i,1) + dt / 2.0, y(i,:) + dt * f2' / 2.0 );
    f4 = dydt ( t(i,1) + dt,       y(i,:) + dt * f3' );

    t(i+1,1) = t(i,1) + dt;
    y(i+1,:) = y(i,:) + dt * ( f1' + 2.0 * f2' + 2.0 * f3' + f4' ) / 6.0;

  end

  return
end
```

Listing 3: The rk4 ODE solver.

Unless you are very patient and careful, I suggest you copy rk4.m from the web site, rather than trying to type it in!

As we saw in the *rk2.m* source code, the right hand side values are column vectors, and have to be transposed with an apostrophe before being used in computations involving the row vectors containing solution values.

5 *expsin_solution.m*, a new test ODE

So far we have judged the improved accuracy of Runge Kutta methods by looking at plots. Now it's time to get seriously numerical. To do so, we'll consider a simple test function called *expsin*, whose formula is

$$y(t) = e^{\sin(t)}$$

Write a corresponding MATLAB function *expsin_solution.m* **with the structure**:

```
function value = expsin_solution ( t )
  value = ?;
  return
end
```

Listing 4: Pseudocode for expsin function.

Plot this function over the interval $0 \leq t \leq 10$, and observe that it has a simple periodic behavior.

6 *expsin_deriv.m*, the derivative function

Determine the formula for $y'(t)$.

Write a corresponding MATLAB function *expsin_deriv.m* **with the structure:**

```
function value = expsin_deriv ( t, y )
  value = ?;
  return
end
```

Listing 5: Pseudocode for expsin ODE right hand side.

7 *expsin_errors.m*, solve the expsin ODE 3 ways

You will need to write a MATLAB function *expsin_errors.m* which solves the *expsin* problem. You are going to solve it using each of the methods *rk1*, *rk2*, *rk4*, and for each method you are going to use several values of the number of steps, `n=5, 10, 20, 40, 80`. It turns out you could do this by writing a program that does all 15 computations, or a program that does one particular computation, which you have to modify for each new computation.

Solve the problem over the interval $0 \leq t \leq 10$, using an initial value of `y0=1.0`.

Each time that you compute the approximate solution, you must also use *expsin_solution(t)* to compute the exact solution at the same points, and use *rms.m* to compute the RMS norm of the difference between the approximate and exact solutions.

```
function expsin_errors ( n )
  dydt = ?
  t0 = ?
  tstop = ?
  tspan = ?
  y0 = ?

  [ t, y1 ] = rk1 ( ? );

  y = ?

  e = rms ( y - y1 );

  fprintf ( 1, '  n = %d, ||y-y1|| = %g\n', n, e );

  return
end
```

Listing 6: An output of expsin_errors.m for rk1

Write the MATLAB code necessary to solve the *expsin* ODE with each of the three Runge-Kutta methods, and the several values of `n`.

Thus, you will be filling out the entries in the following table, where `y1`, `y2` and `y4` are the approximate solutions returns by *rk1*, *rk2* and *rk4*, and `y` is the exact solution vector.

```
         rk1          rk2          rk4
         rms(y-y1)    rms(y-y2)    rms(y-y4)
n =  5:  .........    .........    .........
n = 10:  .........    .........    .........
n = 20:  .........    .........    .........
n = 40:  .........    .........    .........
n = 80:  .........    .........    .........
```

I expect that you will observe several things in this table:

- for a given `n`, the RMS error decreases reading from left to right (increasing Runge-Kutta order);
- for a given Runge Kutta solver, the RMS error decreases as we read from top to bottom (increase `n`;
- for a given Runge Kutta solver, doubling `n` divides the error roughly by 2, 4, or 16;

8 Homework #4

Submit a copy of your completed table to `trenchea@pitt.edu`