The Midpoint and Runge Kutta Methods

http://people.sc.fsu.edu/~jburkardt/isc/week10_lecture_19.pdf

.....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 13 July 2011

The Midpoint and Runge Kutta Methods

- **Introduction**
- The Midpoint Method
- A Function for the Midpoint Method
- Solving Multiple Equations
- Solving A Second Order Equation
- Runge Kutta Methods
- Assignment #8

INTRO: Schedule

Next Class:

- Image Processing

Assignment:

- Programming Assignment #7 is due today.
- Programming Assignment #8 will be due July 21.

INTRO: More Methods, More Problems

We considered a special kind of differential equation called an *initial value problem*, in which we were given a starting time t_0, the value of a quantity at that time, u_0, and a rule for how the quantity changed over time, involving a derivative:

$$\frac{du}{dt} = f(t, u)$$

and we were asked to "solve" this problem.

We agreed that a computational solution would simply be a series of pairs of values $(t_1, u_1), (t_2, u_2), \ldots$ which we think of as points along a solution curve.

INTRO: More Methods, More Problems

To produce a computational solution, we used the Euler method, which essentially uses the derivative information to make a linear prediction about the value at the next desired time. But this method could be inaccurate, and improving accuracy could require taking very many small steps.

Thus, we would like to find some alternatives to the Euler method that are more accurate.

We would also like to know how we can solve problems in which two or three variables change together, a coupled system, or in which the rule of change does not involve the first derivative, but instead $\frac{d^2u}{dt^2}$, a second order system.

INTRO: Using a Library

When we get to the Runge Kutta method, I will show you a little bit about how to use a library. That is, we suppose that someone has written some C++ code that is useful to you. Instead of copying that code into your program, you leave it as a separate file, and tell the compiler to combine the two when it is time to create an executable program.

There are some simple rules for using a library that you should be familiar with. Libraries make it possible for you to write a very small main program that takes advantage of powerful algorithms you didn't have to write yourself.

The Midpoint and Runge Kutta Methods

- Introduction
- **The Midpoint Method**
- A Function for the Midpoint Method
- More Example Differential Equations
- Solving Multiple Equations
- Solving A Second Order Equation
- Runge Kutta Methods
- Assignment #8

MIDPOINT:

One way to think about Euler's method is that it uses the derivative at the current solution point (t_0, u_0) to create a linear prediction of the solution at the next time t_1.

This is like trying to draw a curve by using a sequence of straight line segments. It will never be exact, and a good approximation can require very short line segments, especially if the curve is "curvy".

The midpoint method tries for an improved prediction. It does this by taking an initial half step in time, sampling the derivative there, and then using that forward information as the slope. In other words, it replaces the tangent line by a line that is starting to bend correctly.

MIDPOINT:

The reason that the midpoint rule is better requires working out the Taylor series approximation to the solution. We'll take this on faith. We'll concentrate, instead, on the rules for producing this better estimate of the next solution:

MIDPOINT:

A step of the midpoint method begins with **(t0,u0)** and a stepsize **dt**.

Half Step:

```
th = t0 + dt / 2;
uh = u0 + ( dt / 2 ) * f ( t0, u0 );
```

Full Step:

```
t1 = t0 + dt;
u1 = u0 + dt * f ( th, uh );
```

<--- *f is evaluated "half way"*

The half step essentially peeks into the future, and tries to guess which way the wind is blowing midway between the current and future times.

The Midpoint and Runge Kutta Methods

- Introduction
- The Midpoint Method
- **A Function for the Midpoint Method**
- Solving Multiple Equations
- Solving A Second Order Equation
- Runge Kutta Methods
- Assignment #8

MIDPOINT.CPP:

Fortunately, the midpoint method uses the same input and output as Euler's method. Therefore, we need:

- **t0**, the starting time;
- **u0**, the starting value;
- **dt**, the stepsize;
- **f(t,u)**, a function that evaluates the derivative.

and what emerges as the function value is **u1**, the approximate solution at time $t1$.

So our function will be declared as

```
double midpoint ( double t0, double u0, double dt,
  double f ( double t, double u ) );
```

MIDPOINT.CPP: The Midpoint Code

The midpoint method is wordier than Euler was:

```
double midpoint ( double t0, double u0, double dt,
  double f ( double t, double u ) )
{
  double th, uh, u1;
//
//  Take a half time step and estimate UH there.
//
  th = t0 + 0.5 * dt;
  uh = u0 + 0.5 * dt * f ( t0, u0 );
//
//  Evaluate the derivative at (TH,UH).
//
  u1 = u0 +       dt * f ( th, uh );

  return u1;
}
```

MIDPOINT_F3.CPP: A Program For The Wiggly Equation

Let's try to integrate the wiggly function:

```
# include <cstdlib>
# include <iostream>
using namespace std;

double midpoint ( double t0, double u0, double dt, double f ( double t, double u ) );
double f3 ( double t, double u );

int main ( )
{
  double dt = 1.0, exact, pi = 3.14159265, t0, t1, tmax, u0, u1;  <-- Initial data

  u0 = 0.5;
  t0 = 0.0;
  tmax = 12.0 * pi;
  dt = 0.1;

  while ( true )
  {
    cout << t0 << "  " << u0 << "\n";
    if ( tmax <= t0 )                                  <-- Did we reach our goal?
    {
      break;
    }
    t1 = t0 + dt;                                      <-- Take another step.
    u1 = midpoint ( t0, u0, dt, f3 );

    t0 = t1;                                           <-- Shift data for next loop.
    u0 = u1;
  }
  return 0;
}
```

...Text of "midpoint.cpp" and "f3.cpp" follows...

ERROR: Comparing Approximate Solutions

When we run our **midpoint_f3.cpp** program for different time steps, we concentrate on a few selected points:

	<---------EULER------------>			<---------MIDPOINT--------->		
T	U	U	U	U	U	U
	0.25	0.10	0.01	0.25	0.10	0.01
0.0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
10.0	0.185518	0.244324	0.285323	0.292115	0.290496	0.290208
20.0	0.363628	0.769275	1.187460	1.295780	1.246530	1.245830
30.0	0.029936	0.091483	0.173494	0.189469	0.186583	0.186158

For the midpoint method, our estimated solution values don't change drastically as we decrease the step size. This suggests our midpoint calculation is more accurate.

ERROR: Comparing Approximate Solutions

Normally, comparing solutions for different stepsizes is the best we can do. Since I happen to know the correct solution for this problem, let me show the errors for the midpoint method now:

	<---------MIDPOINT--------->					
	<---------SOLUTION--------->			<----------ERRORS---------->		
T	U	U	U			
	0.25	0.10	0.01	0.25	0.10	0.01
0.0	0.500000	0.500000	0.500000	0.0	0.0	0.0
10.0	0.292115	0.290496	0.290208	0.0019	0.00029	0.0000027
20.0	1.295780	1.246530	1.245830	0.0071	0.00070	0.0000043
30.0	0.189469	0.186583	0.186158	0.0033	0.00042	0.0000036

For the midpoint method, dividing the stepsize by 10 reduces the error by 100. This is called a "quadratic" error reduction, and means we can get accurate solutions faster.

The Midpoint and Runge Kutta Methods

- Introduction
- The Midpoint Method
- A Function for the Midpoint Method
- **Solving Multiple Equations**
- Solving A Second Order Equation
- Runge Kutta Methods
- Assignment #8

MULTIPLE: Example 4: Predator/Prey

Example 4: Predator/Prey

On Survivor Island, we have two animal populations, rabbits, and foxes. The rabbits survive by munching grass, but the foxes munch on rabbits.

Let r represent the number of rabbits, and f the number of foxes, which are really functions of time that we may also write as $r(t)$ and $f(t)$.

The chance that a particular rabbit will meet a particular fox in a unit of time has been measured as the number α. Therefore, the chance that some rabbit will meet some fox is $\alpha * r * f$. Meetings like this decrease the number of rabbits, and give the foxes a means to have baby foxes.

Without these encounters, the rabbits simply reproduce, at a rat $2r$, and the foxes gradually die out, at a rate of $-f$.

MULTIPLE: Example 4: Predator Prey

These equations describe a predator-prey system of equations:

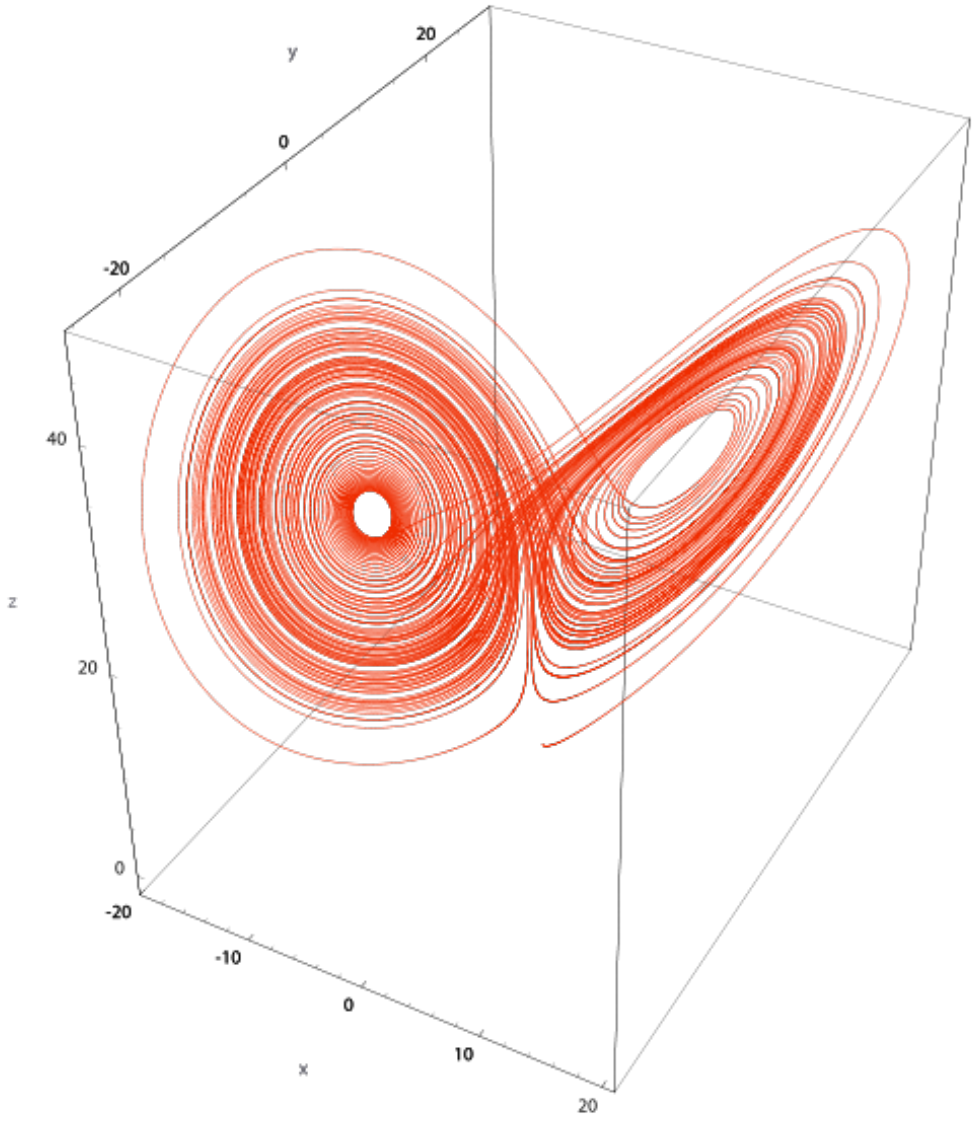
$$\frac{dr}{dt} = 2r - \alpha\, r\, f$$
$$\frac{df}{dt} = -f + \alpha\, r\, f$$

with r_0 and f_0 specified at time $t = 0$.

A sample set of data is $\alpha = 0.01$, $r_0 = 300$, $f_0 = 150$.

Notice that these equations are *coupled* - that is, we can't solve the equation for $r(t)$ without working out the solution for $f(t)$ as well.

MULTIPLE: Example 5: The Lorenz Equations



MULTIPLE: Example 5: The Lorenz Equations

Edward Lorenz was trying to set up an extremely simplified model of weather, and came up with the following system of equations:

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= \rho x - y - xz \\ \frac{dz}{dt} &= xy - \beta z\end{aligned}$$

where $\sigma = 10$, $\beta = \frac{8}{3}$ and the value of ρ varies, but often chosen to be 28.

These equations are also coupled, but there is more going on th that! The answer is very sensitive to the initial values.

MULTIPLE: Coupled Differential Equations

We said earlier that differential equations are used to explain how a system changes. Sometimes, the system we are interested in has several quantities, which affect each other. In that case, it may be that the system can be modeled by a set of several differential equations which are coupled.

The methods we have studied for a single differential equation can easily be extended to this case. However, in C++, we will have to move from dealing with scalar variables to using arrays.

To keep things simple, we will go back to the Euler method, and consider how it can be adapted to handle the predator prey problem.

MULTIPLE: The Solution is an Array

We will still think of the variable name **u** as describing the solution variable, but now **u** must be an array of dimension 2.

Thus, the initial condition for the predator prey problem is set by

```
double dt = 0.1, t0 = 0.0, u0[2] = { 300, 150 };
```

Similarly, we can set up a second array **u1[2]**, to hold the next value of the solution.

For convenience, we can copy entries of the array and call them **r** and **f**, but the storage and update of information should generally be done with arrays.

MULTIPLE: The Derivative Function

It will be easiest to change the derivative function **f()** so that the derivative array is an argument **dudt[]**, rather than being returned as the function value. For the predator prey problem, we have:

```
void f4 ( double t, double u[], double dudt[] )
{
  double alpha = 0.01, f, r;

  r = u[0];
  f = u[1];

  dudt[0] = 2 * r - alpha * r * f;
  dudt[1] =   - f + alpha * r * f;

  return;
}
```

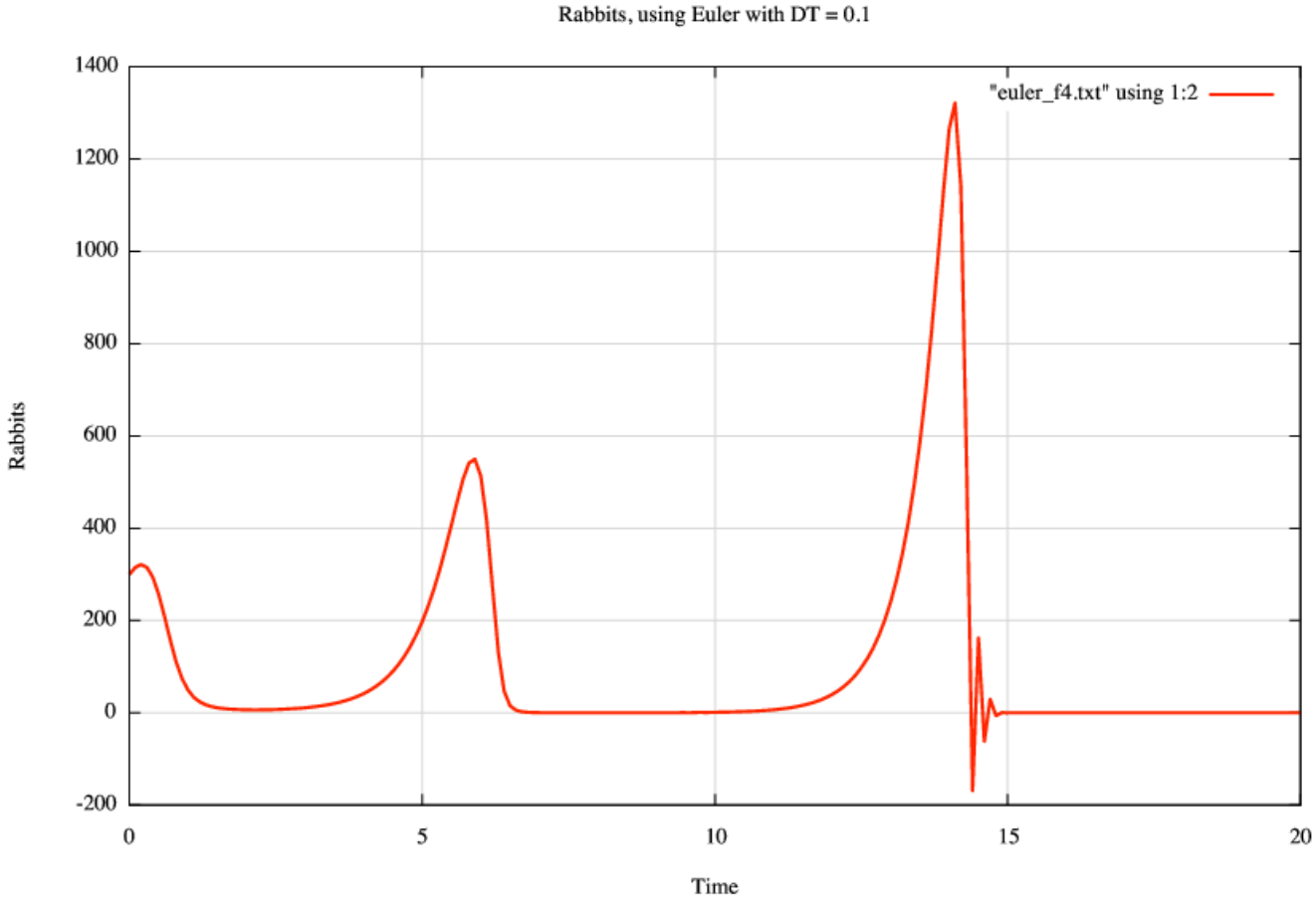
MULTIPLE: The Derivative Function

Trying to set up an Euler function when arrays are involved is a little too complicated for our beginning class. Instead, let's bring the Euler computation into the main program.

```
  while ( true )
  {
    cout << "  " << t0 << "  " << u0[0] << "  " << u0[1] << "\n";
    if ( tmax <= t0 )
    {
      break;
    }
//
//  New derivative vector.
//
    f ( t0, u0, dudt );
//
//  Update T1, U1 using the Euler method.
//
    t1 = t0 + dt;
    for ( i = 0; i < 2; i++ )
    {
      u1[i] = u0[i] + dt * dudt[i];
    }
//
//  Shift data for next loop.
//
    t0 = t1;
    for ( i = 0; i < 2; i++ )
    {
      u0[i] = u1[i];
    }
  }
```

MULTIPLE: Rabbits using Euler with DT = 0.1

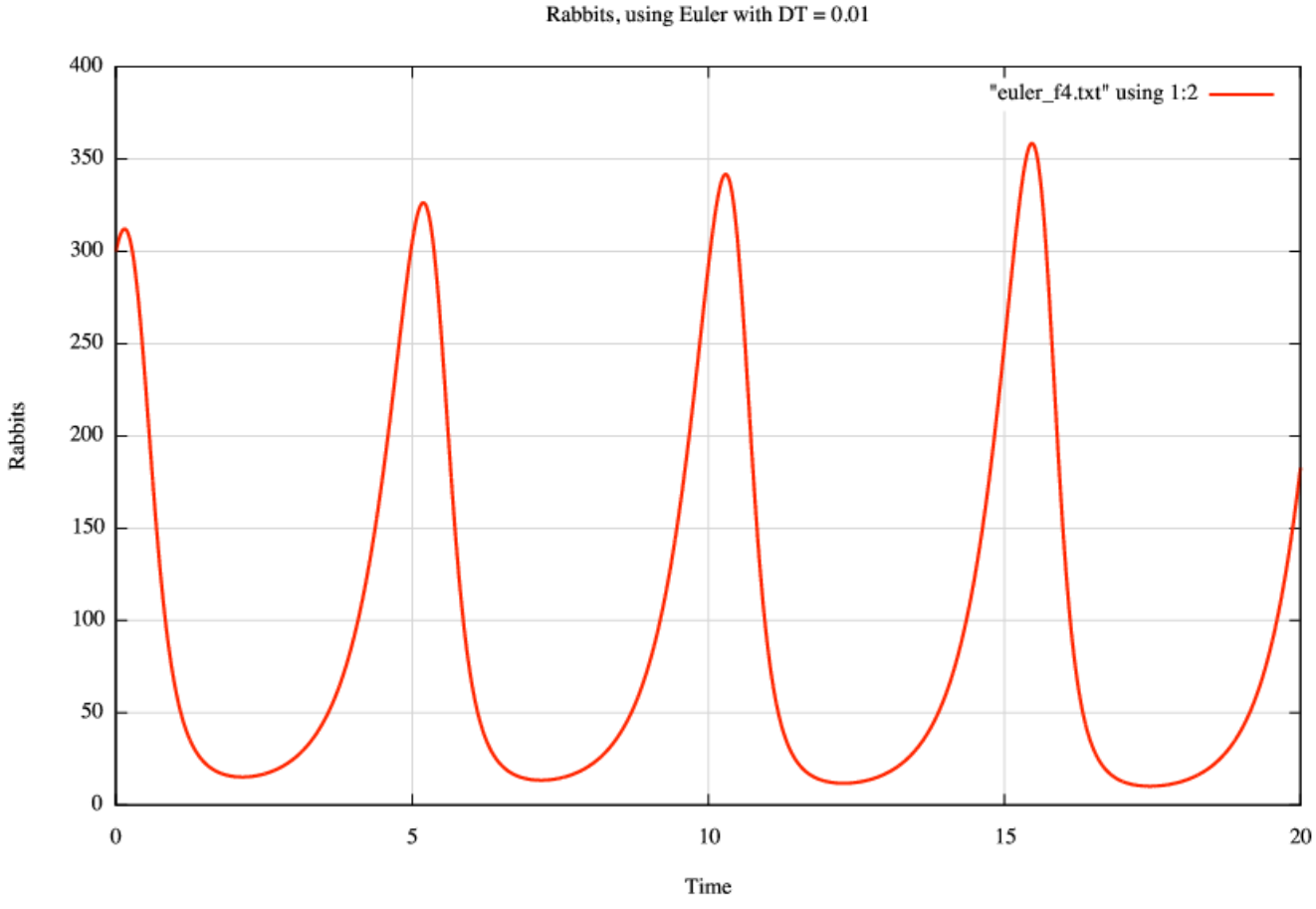
The Euler method with DT = 0.1 computes negative rabbits!




```
plot "euler_f4.txt" using 1:2 with lines
```

MULTIPLE: Rabbits using Euler with DT = 0.01

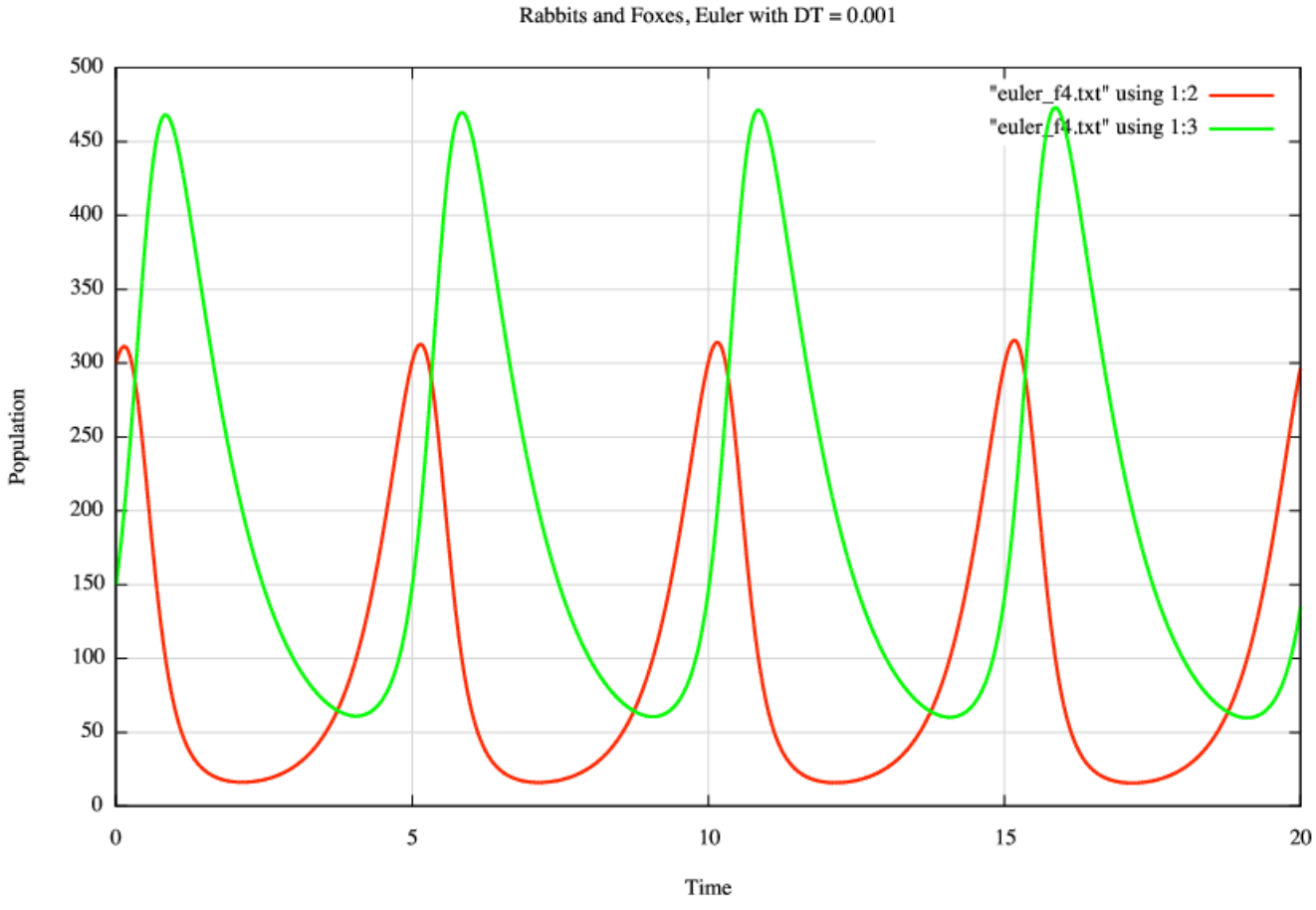
With DT = 0.01, results are better, but they grow.




```
plot "euler_f4.txt" using 1:2 with lines
```

MULTIPLE: Rabbits and Foxes, Euler with DT = 0.001

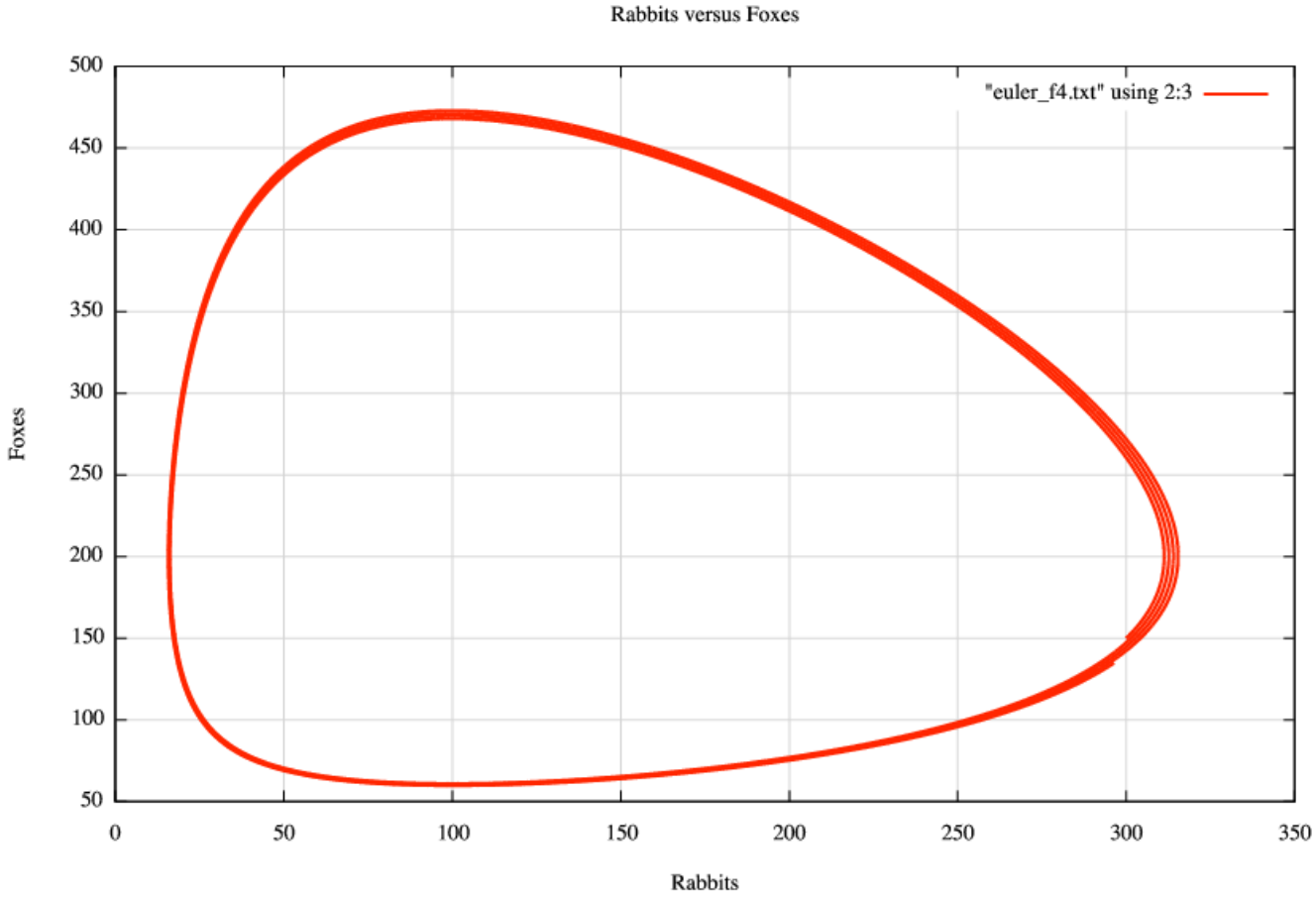
With DT = 0.001, results are almost periodic.




```
plot "euler_f4.txt" using 1:2 with lines,
     "euler_f4.txt" using 1:3 with lines
```

MULTIPLE: Phase Plane

Plotting rabbits versus foxes shows us the cyclic behavior.




```
plot "euler_f4.txt" using 2:3 with lines
```

The Midpoint and Runge Kutta Methods

- Introduction
- The Midpoint Method
- A Function for the Midpoint Method
- Solving Multiple Equations
- **Solving A Second Order Equation**
- Runge Kutta Methods
- Assignment #8

SECOND: Example 6: The Cannon Equation

SECOND: Example 6: The Cannon Equation

A cannon is aimed at angle α, and charged with enough powder that the cannonball exits the cannon with speed s.

Thus, at time $t_0 = 0$, we have initial position $\vec{p}(t_0) = (x, y) = (0, 0)$ and velocity $\vec{v}(t_0) = (s * \cos(\alpha), s * \sin(\alpha))$.

The subsequent position of the cannonball satisfies

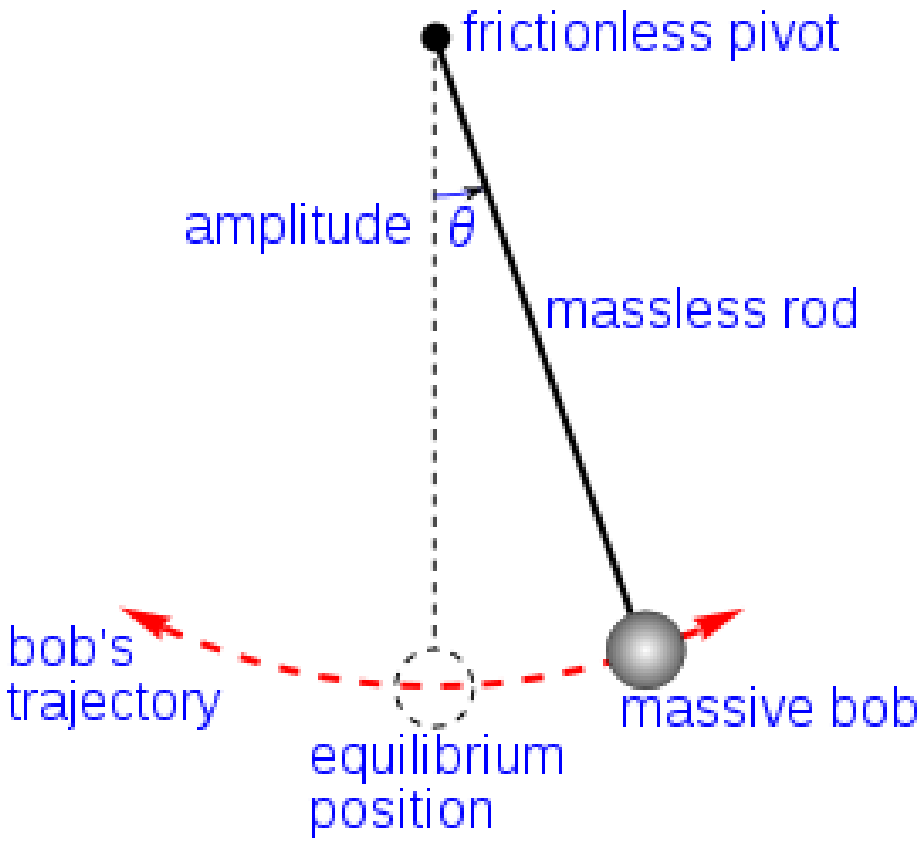
$$\frac{d^2x}{dt^2} = 0$$
$$\frac{d^2y}{dt^2} = -g$$

SECOND: Example 6: The Cannon Equation

Typical data would have $s = 50\frac{meter}{sec}$, $\alpha = 30^o = \frac{\pi}{6}$ and g is $32\frac{feet}{sec^2}$ or $9.8\ \frac{meters}{sec^2}$.

This is a pair of equations, and both equations are second order, that is, they involve the second derivative of the quantity of interest.

SECOND: Example 7: The Pendulum



SECOND: Example 7: The Pendulum

The angle θ made by a pendulum satisfies the equation:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta)$$

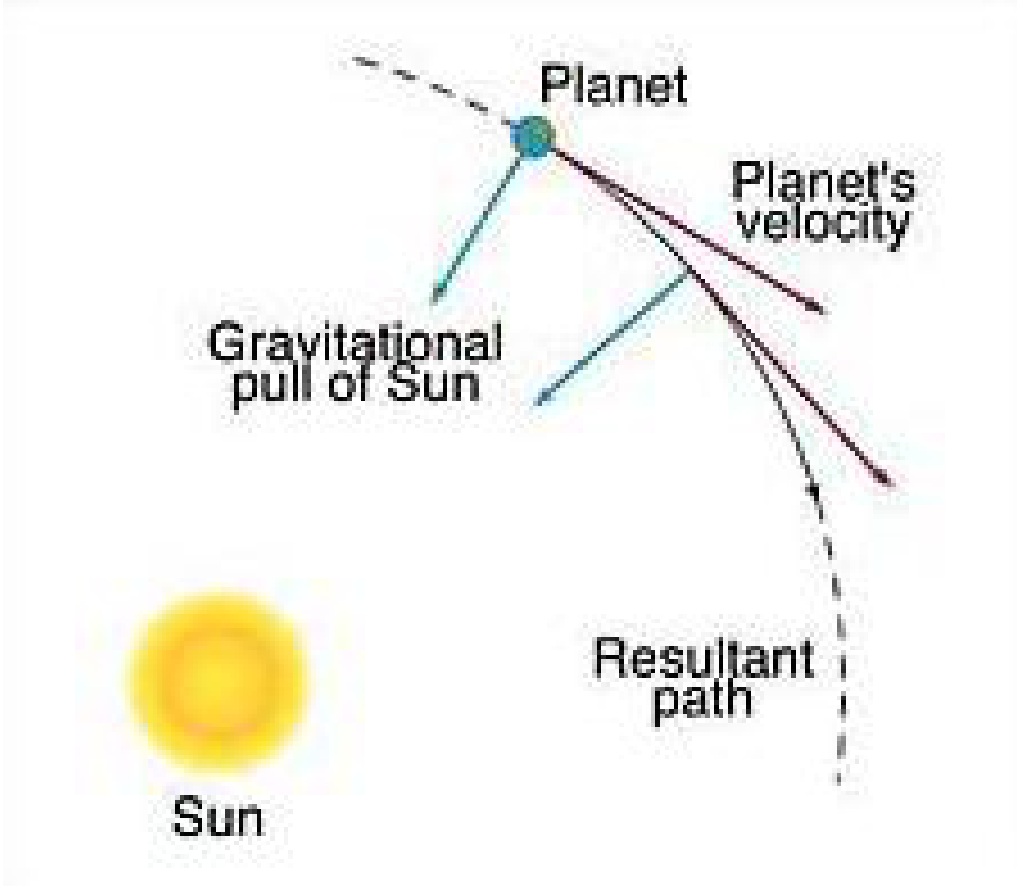
where g is the force of gravity, l is the length of the pendulum, and θ is the angle the pendulum makes from the downward vertical vector.

At the starting time t_0, we are given the initial angle $\theta(t_0)$ and the angular velocity, $\frac{d\theta}{dt}$.

Typical initial data is $t_0 = 0$, $\theta(t_0) = \frac{\pi}{4}$, and $\frac{d\theta}{dt}(t_0) = 0$, with g = 386.09 $\frac{in}{sec^2}$ and l = 10 inches.

This problem involves a second derivative.

SECOND: Example 8: Planetary Motion



SECOND: Example 8: Planetary Motion

As a function of time, the position vector $p(t)$ of a planet satisfies the vector equation

$$\frac{d^2\vec{p}}{dt^2} = -\frac{\vec{p}}{||\vec{p}||^3}$$

or, writing $p(\vec{t}) = (x(t), y(t))$

$$\frac{d^2x}{dt^2} = \frac{-x}{(x^2+y^2)^{\frac{3}{2}}}$$
$$\frac{d^2y}{dt^2} = \frac{-y}{(x^2+y^2)^{\frac{3}{2}}}$$

This is a pair of coupled, second order equations.

SECOND: Rewrite as a System

We know how to solve an initial value problem of the form $\frac{du}{dt} = f(t, u)$; we can even work with problems where u is really an array of several values that change together.

Nothing has prepared us for a problem involving a second derivative. But since we have a great hammer, we want everything to look like a nail...so let's try to transform the second order differential equation.

SECOND: Rewrite as a System

Let's write a typical second order equation as:

$$\frac{d^2v}{dt^2} = f(t, v)$$

Now let's make a vector of new variables, with the property that

$$\begin{aligned} u[0] &= v \\ u[1] &= \frac{dv}{dt} \end{aligned}$$

What happens if we differentiate these two equations?

$$\begin{aligned} \frac{du[0]}{dt} &= \frac{dv}{dt} = u[1] \\ \frac{du[1]}{dt} &= \frac{d^2v}{dt^2} = f(t, v) = f(t, u) \end{aligned}$$

SECOND: Rewrite as a System

In other words, a second order equation like:

$$\frac{d^2v}{dt^2} = f(t, v)$$

can always be rewritten as a pair of first order equations:

$$\begin{aligned}\frac{du[0]}{dt} &= u[1] \\ \frac{du[1]}{dt} &= f(t, u)\end{aligned}$$

and as long as we have initial conditions for both $\mathbf{v}$ and $\frac{dv}{dt}$, we can solve these problems too!

SECOND: The Pendulum Equation

The pendulum equation looks like:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin(\theta)$$

but we can replace this by a pair of equations:

$$\begin{aligned}\frac{du[0]}{dt} &= u[1] \\ \frac{du[1]}{dt} &= -\frac{g}{l}\sin(u[0])\end{aligned}$$

SECOND: The Pendulum Equation

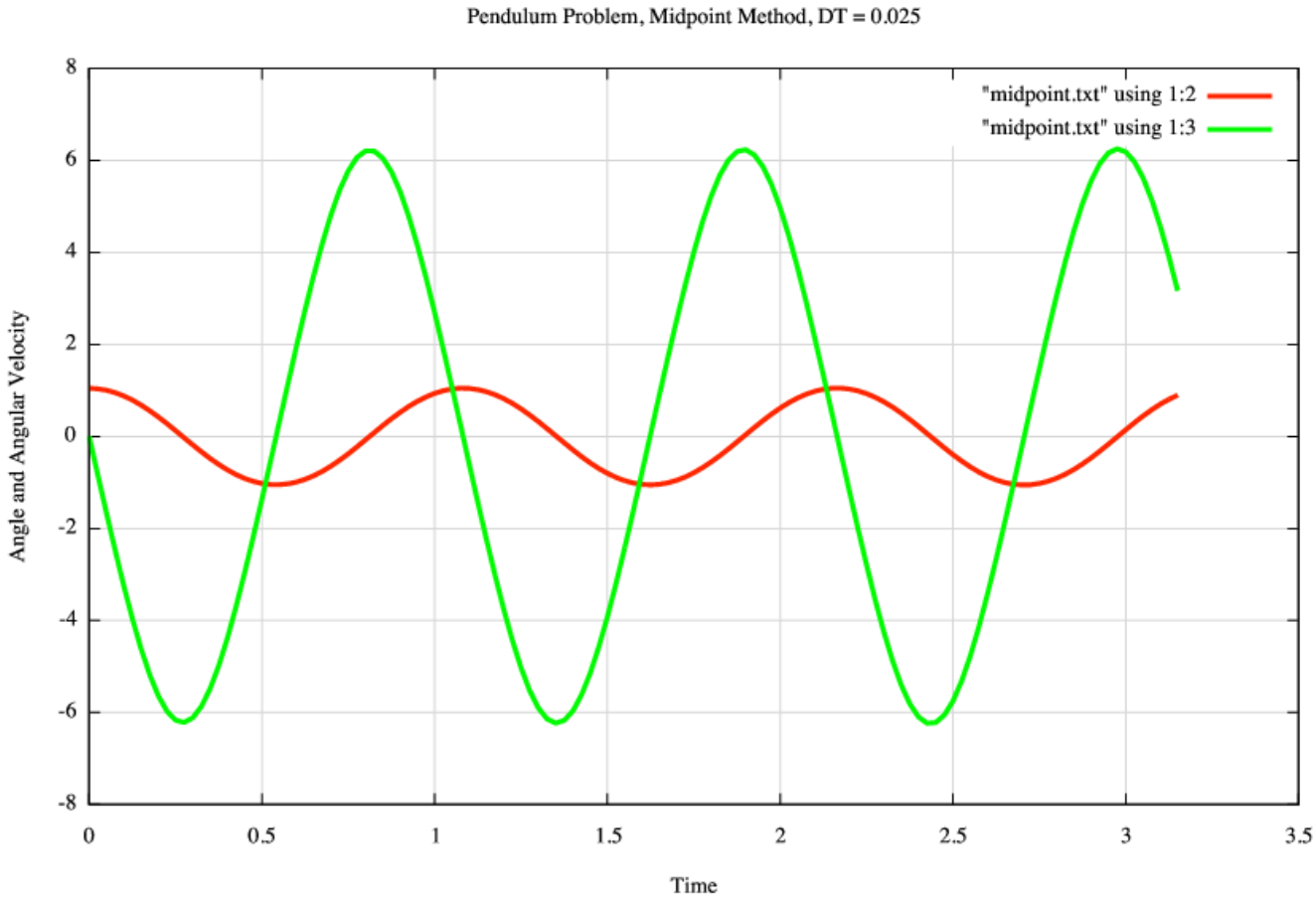
So the derivative function would be:

```
void f7 ( double t, double u[], double dudt[] )
{
  double g = 386.09, l = 10.0;

  dudt[0] = u[1];
  dudt[1] = - ( g / l ) * sin ( u[0] );

  return;
}
```

SECOND: The Midpoint Time Loop

```
  while ( true )
  {
    cout << "  " << t0 << "  " << u0[0] << "  " << u0[1] << "\n";
    if ( tmax <= t0 )
    {
      break;
    }
//  Half step.

    f7 ( t0, u0, dudt );
    th = t0 + 0.5 * dt;
    for ( i = 0; i < 2; i++ )
    {
      uh[i] = u0[i] + 0.5 * dt * dudt[i];
    }
//  Full step.

    f7 ( th, uh, dudt );
    t1 = t0 + dt;
    for ( i = 0; i < 2; i++ )
    {
      u1[i] = u0[i] + dt * dudt[i];
    }
//  Shift.

    t0 = t1;
    for ( i = 0; i < 2; i++ )
    {
      u0[i] = u1[i];
    }
  }
```

SECOND: Plot of Angle and Angular Velocity




```
plot "midpoint_f7.txt" using 1:2 with lines,
     "midpoint_f7.txt" using 1:3 with lines
```

SECOND: Example 6: The Cannon Equation

The cannon equation involves two equations, both of second order. To handle this, create a single array **u[]** of size 4, setting:

$$u[0] = x, \quad u[1] = \frac{dx}{dt},$$
$$u[2] = y, \quad u[3] = \frac{dy}{dt}$$

so the equations become:

$$\frac{du[0]}{dt} = u[1]$$
$$\frac{du[1]}{dt} = 0$$
$$\frac{du[2]}{dt} = u[3]$$
$$\frac{du[3]}{dt} = -g$$

and now set up the system of four equations.

The Midpoint and Runge Kutta Methods

- Introduction
- The Midpoint Method
- A Function for the Midpoint Method
- Solving Multiple Equations
- Solving A Second Order Equation
- **Runge Kutta Methods**
- Assignment #8

RK4:

If you a solving a problem in C++, there's always a simple technique, like the Euler or midpoint method, that might work for you. But sometimes your problem is more difficult or you need a better guarantee of accuracy. At that time, the smart thing to do is to hope that someone else has already written a piece of C++ code for your problem

I would like to show you how you do this. We will do this by trying to find a better solver for the "wiggly" function. The solver we will choose is called the *Runge Kutta method of order 4* or **RK4**, and just from the name, you can guess you probably don't want to have to write this code yourself!

When people write useful codes, they put them in libraries for others to use. We'll assume we've found a library with the **RK4** solver in it, and try to use it.

RK4:

The **RK4** solver is available to us in a library. We happen to have access to the *source code*, that is, the text of the C++ file. Often, you only get a copy of the compiled code. Since a library is not a complete program, the compiler has to be warned not to try to make a program out of it. This is done using the extra switch **-c**.

```
g++ -c rk4.cpp
```

In that case, the compiler creates what is called an "object file", in this case, the file **rk4.o**. Even if the source code is no longer available, the object file can be used to build an executable program.

RK4:

Since we didn't write the **rk4.cpp** file, we need to know how to call the **rk4()** function it contains. Luckily, we have a copy of the source code; otherwise, we'd have to hope there was a manual, or some online information.

The declaration looks just like the one for the simple Euler method:

```
double rk4 ( double t0, double u0, double dt,
  double f ( double t, double u ) )
```

and the code works the same way.

RK4:

Since we are going to call the **rk4()** function, we must include the declaration of that function in our main program:

```
double rk4 ( double t0, double u0, double dt,
  double f ( double t, double u ) );
```

except that, like most libraries, the author has made an extra "include" file available which declares all the things in the library. We use it by inserting the statement:

```
# include "rk4.hpp"
```

at the beginning of the program with all the other **include** statements.

RK4:

So, once we've declared **rk4()** using an **include** statement, and replaced the call to **euler** by a call to **rk4()**, we're almost ready.

Now, we need to compile our main program, and tell the compiler to add on the library code as well:

```
g++ myprog.cpp rk4.o  <-- rk4.cpp was compiled earlier
```

and the compiler will create the usual **a.out** executable file.

Libraries usually come already compiled. If our program simply came in two pieces, we could compile them at the same time with the command

```
g++ myprog.cpp rk4.cpp
```

RK4:

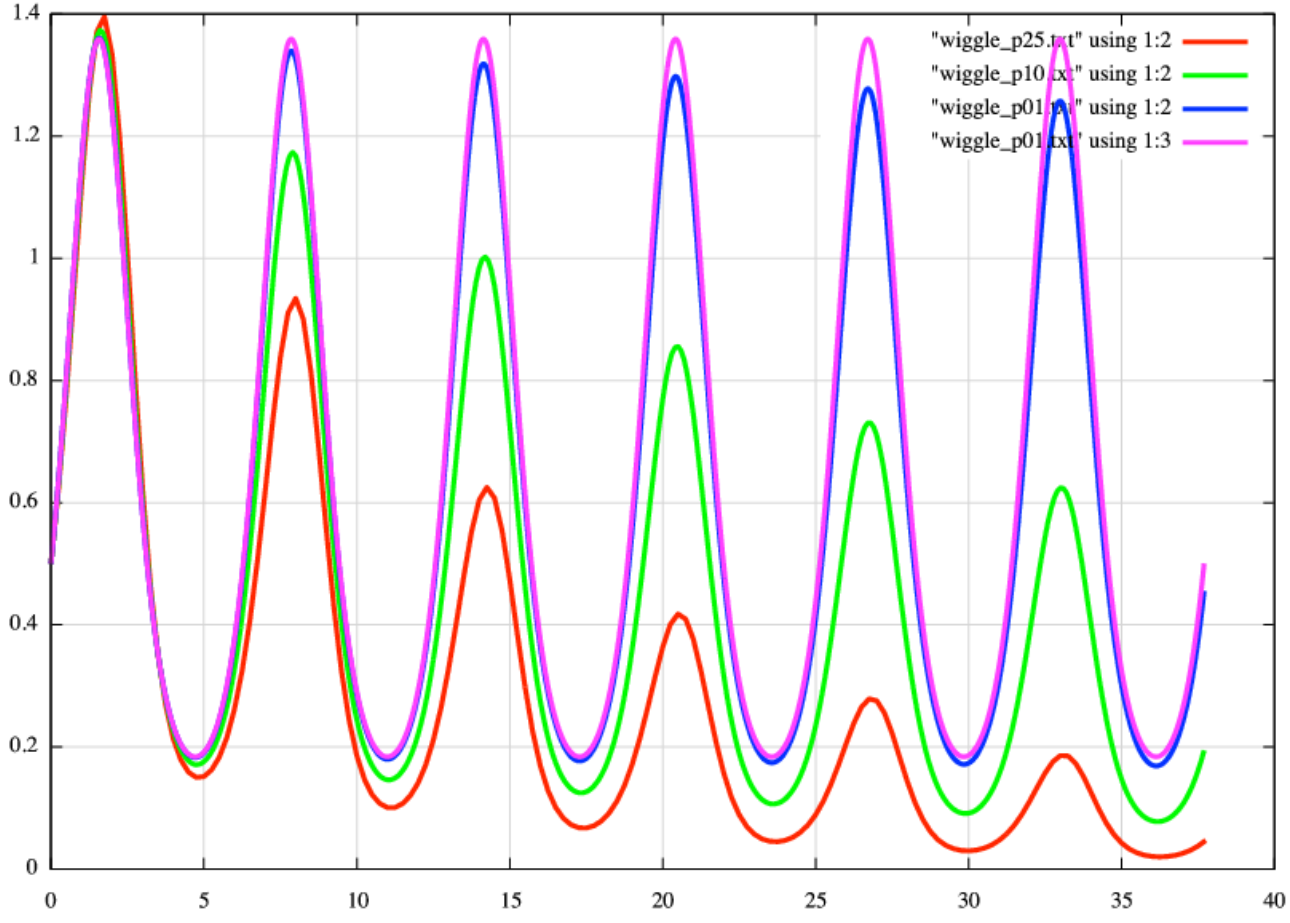
Our main program looks like this:

```
# include <cstdlib>
# include <iostream>
# include <cmath>
using namespace std;

# include "rk4.hpp"

double f3 ( double t, double u );

int main ( )
{
  double dt = 0.1, pi = 3.14159265, t0 = 0.0, t1;
  double tmax = 12.0 * pi, u0 = 0.5, u1;

  while ( true )
  {
    cout << "  " << t0 << "  " << u0 << "\n";
    if ( tmax <= t0 )
    {
      break;
    }
//  Advance to time T1.
    t1 = t0 + dt;
    u1 = rk4 ( t0, u0, dt, f3 );
//  Shift the data.
    t0 = t1;
    u0 = u1;
  }
  return 0;
}
```

...plus text of f3() function...

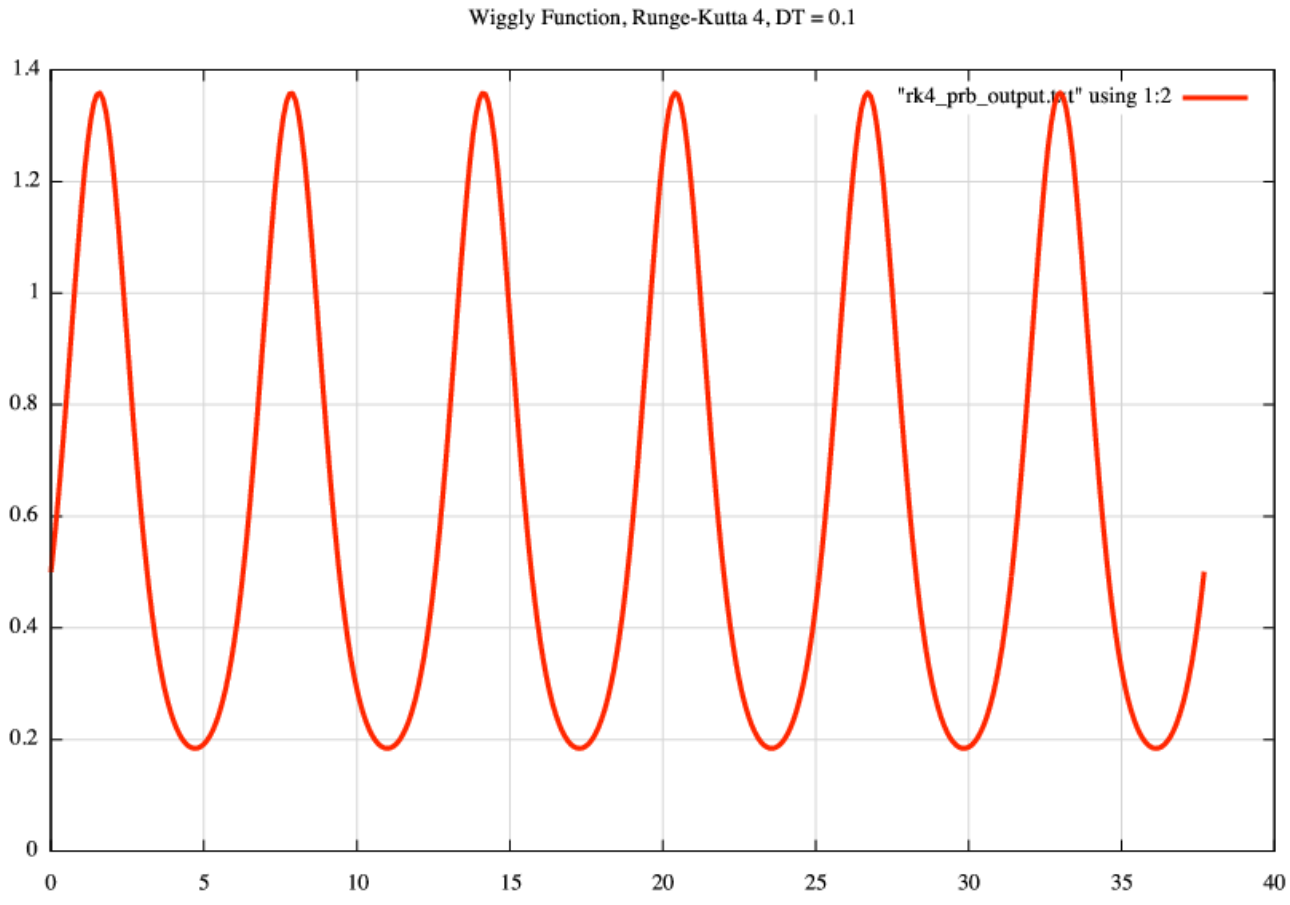
RK4:

Here is how Euler worked with steps as small as DT = 0.01.



RK4:

Here is how the RK4 function worked with a step size DT = 0.1:



The Midpoint and Runge Kutta Methods

- Introduction
- The Midpoint Method
- A Function for the Midpoint Method
- More Example Differential Equations
- Solving Multiple Equations
- Solving A Second Order Equation
- Runge Kutta Methods
- **Assignment #8**

ASSIGNMENT #8: A Second Order Initial Value Problem

Consider the following ODE:

$$\frac{d^2u}{dt^2} = \frac{u^3}{6} - u + 2 * \sin(2.7853 * t)$$

with initial conditions:

$$\begin{aligned} t_0 &= 0.0 \\ u(t_0) &= 0.0 \\ \frac{du}{dt}(t_0) &= 0.0 \end{aligned}$$

which is to be solved up to $t_{max} = 20$.

ASSIGNMENT #8: A Second Order Initial Value Problem

Rewrite this second order ODE as a pair of first order ODE's. Modify the program **midpoint_f7.cpp** to solve the problem.

The correct solution at $t_{max} = 20$ has the values

$$u(20) = -0.1004...$$

$$\frac{du}{dt}(20) = 0.2411...$$

I won't tell you what stepsize to use. Try to find a value of **dt** small enough so the first three digits of your answer match the correct solution.

Sometimes the program takes one more step than expected, s
sure you report the values corresponding to **t** = 20!

ASSIGNMENT #8: Things to Turn in

Email to Detelina:

- the stepsize **dt** that you used:
- the values $\mathbf{u(20)}$ and $\frac{\mathbf{du}}{\mathbf{dt}}\mathbf{(20)}$ that you computed:
- a copy of your program.

The program and output are due by Thursday, July 21.