The Secant and Newton Methods

http://people.sc.fsu.edu/~jburkardt/isc/week09_lecture_17.pdf

.....

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

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 07 July 2011

The Secant and Newton Methods

- **Introduction**
- Making the Equation a Variable
- Sample Functions
- The Secant Method
- Newton's Method
- Assignment #7

INTRO: Schedule

Next Class:

- Differential Equations

Assignment:

- Programming Assignment #6 is due today.
- A short project proposal is due Tuesday, July 12.
- Programming Assignment #7 will be due July 14.

INTRO: Improving the Bisection Code

In the last lecture, we talked about the bisection method, a simple procedure that slowly squeezes the uncertainty out of an estimate for the solution of a nonlinear equation, assuming we were able to bracket the solution, that is, find a negative and positive function value, so that a crossing must occur in between.

We created a C++ function, **bisect1.cpp**, to carry out this procedure. This meant the user had to write a C++ function to be used by the bisection procedure. And so we had to pick one name for that function and stick with it. We will see a simple way of allowing the user to use any name for the function, just like the user can have any name for the input variables to a function.

INTRO: Methods Faster than Bisection

We will then look at another method for solving nonlinear equations, called the **secant method**, which can be much faster than bisection, but which can fail if we start too far from the solution.

We will then consider a related, but much more powerful solver called **Newton's method**, which uses derivative information to get a more accurate fix on the probable location of the solution. Newton's method is important because it can be modified to handle **systems** of nonlinear equations, that is, two, three or hundreds of equations for which a solution is needed.

The Secant and Newton Methods

- Introduction
- **Making the Equation a Variable**
- Sample Functions
- The Secant Method
- Newton's Method
- Assignment #7

VARIABLE: Must Our Function be Named F(X)?

The bisection program can be very useful, but one of its limitations is that it requires that the equation to be solved is described by a function whose name must be **f()**.

This is inconvenient; we might want our function to be described by some other name.

Moreover, we might have one program that wants to solve several different equations, say **f1()**, **f2()** and **f3()**. Are you saying we can only solve one of these equations, and even for that one we have to rename the function to **f()**?

Surprisingly, we can make our program more flexible, because C++ allows us to use the name of the function to be solved as just another input quantity. In other words, it's easy to tell **bisec()** *"please solve f3(x)=0"*.

VARIABLE: Functions are Similar to Variables

So if we wanted to think of the function we are working with as just another variable, how would we describe it? *We already have!* When we declared function **f1(x)**, we described it as

```
double f1 ( double x );
```

We made similar declarations for functions **f2(x)** and so on. So our bisection solver will work as long as it is given a function, let's call it just plain old **f(x)**, which looks like

```
double f ( double x );
```

So we can add a function name as input to version 2 of the **bisect()** solver:

```
double bisect2 ( double a, double b, double f ( double x ) );
```

VARIABLE:

This means we can have one program that includes our functions **f1()** through **f5()**, and if we want to find a solution of, say, $f3(x) = 0$, we just have to set up the appropriate input to **bisect2()**, namely, the value of **a**, the value of **b**, and the value (that is, the name) of the function **f()**:

```
a = 2.0;
b = 3.0;
c = bisect2 ( a, b, f3 );
```

The program **bisect2_f1f2f3.cpp** is set up this way. Notice that the main program solves equations involving function f1, f2, and f3, but that **bisect2()** treats them all as though they were sim named **f()**.

VARIABLE: BISECT2_F1F2F3.CPP

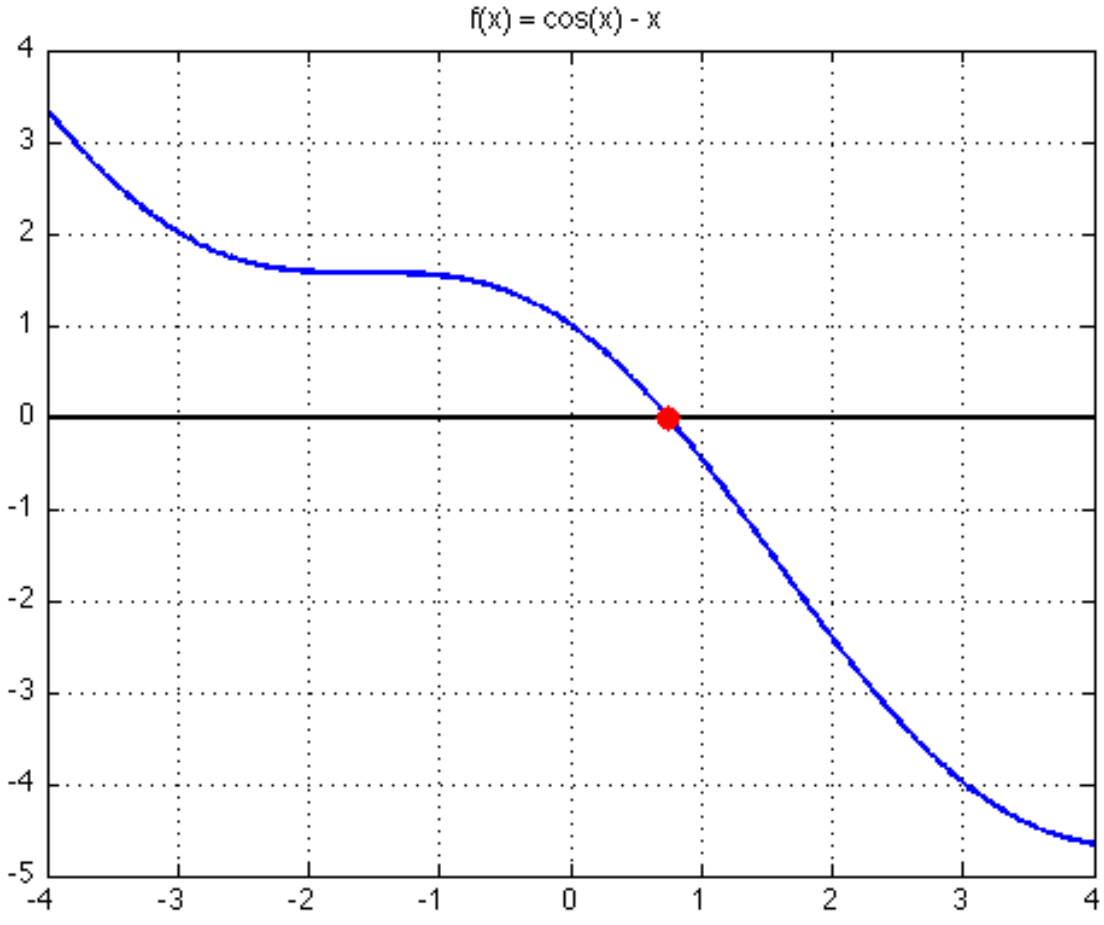
```
double bisect2 ( double a, double b, double f ( double x ) );
double f1 ( double x );
double f2 ( double x );        <--- We declare bisect2(), f1(), f2(), and f3()
double f3 ( double x );             up here, so "everyone" can use them.

int main ( )
{
  double a;
  double b;
  double c;

  a = - 10.0;
  b = + 10.0;
  c = bisect2 ( a, b, f1 );    <-- Solve using function f1()

  cout << "F1(" << C << ") = " << f1 ( c ) << "\n";

  a = 0.0;
  b = 3.0;
  c = bisect2 ( a, b, f2 );    <-- Solve using function f1()

  cout << "F2(" << C << ") = " << f2 ( c ) << "\n";

  a = 2.0;
  b = 3.0;
  c = bisect2 ( a, b, f3 );    <-- Solve using function f1()

  cout << "F3(" << C << ") = " << f3 ( c ) << "\n";

  return 0;
}
```

The Secant and Newton Methods

- Introduction
- Making the Equation a Variable
- **Sample Functions**
- The Secant Method
- Newton's Method
- Assignment #7

FUNCTION 1: The Cosine Equation

Our first function is $f_1(x) = cos(x) - x$:

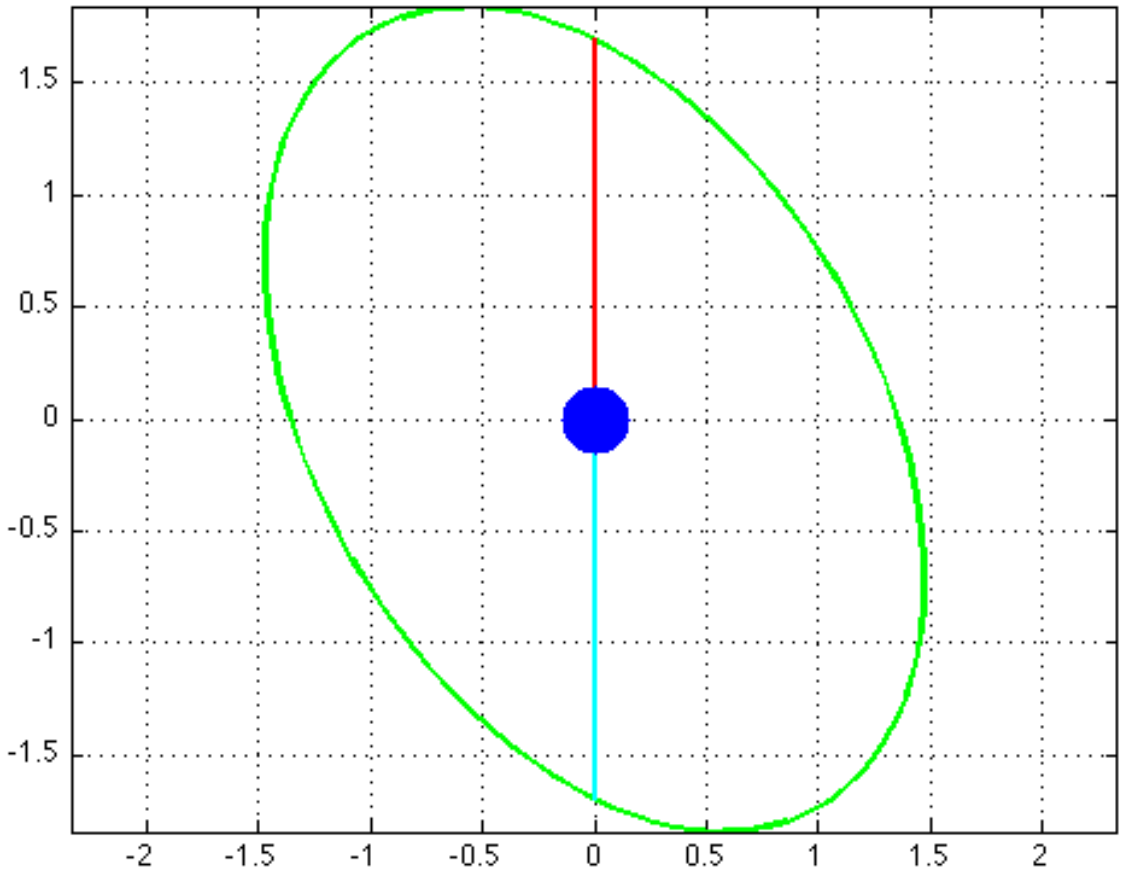


FUNCTION 1: The Cosine Function

```
double f1 ( double x )
{
  double value;

  value = cos ( x ) - x;

  return value;
}
```

FUNCTION 2: The Satellite Equation

When is the satellite directly overhead?



FUNCTION 2: The Satellite Equation

ET is stranded on earth, while his spaceship is circling in an elliptical orbit. ET's only chance is to beam himself up when the spaceship is directly overhead. ET is at coordinates (0,0). The satellite's position, as a function of time, is

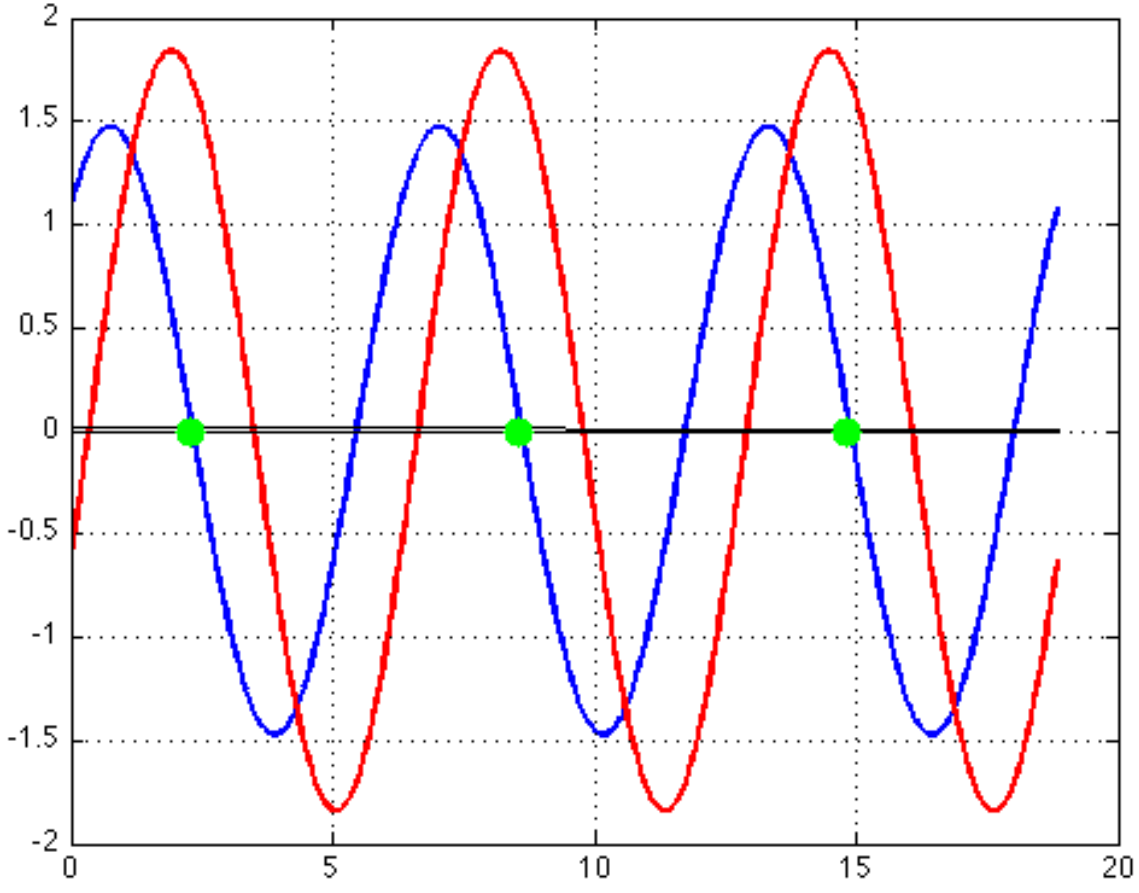
$$x(t) = 1.082 * \cos t + \sin t$$
$$y(t) = -0.625 * \cos t + 1.732 * \sin t$$

The satellite is directly overhead when $x(t)$ is 0 and $y(t)$ is positive. When should ET try to beam up?

So our next function is $f_2(x) = 1.082 * \cos x + \sin x$ which actually has many zeros.

FUNCTION 2: The Satellite Equation

The blue line is our function; the red line must be positive for our solution to be good. That means the green dots are solutions!

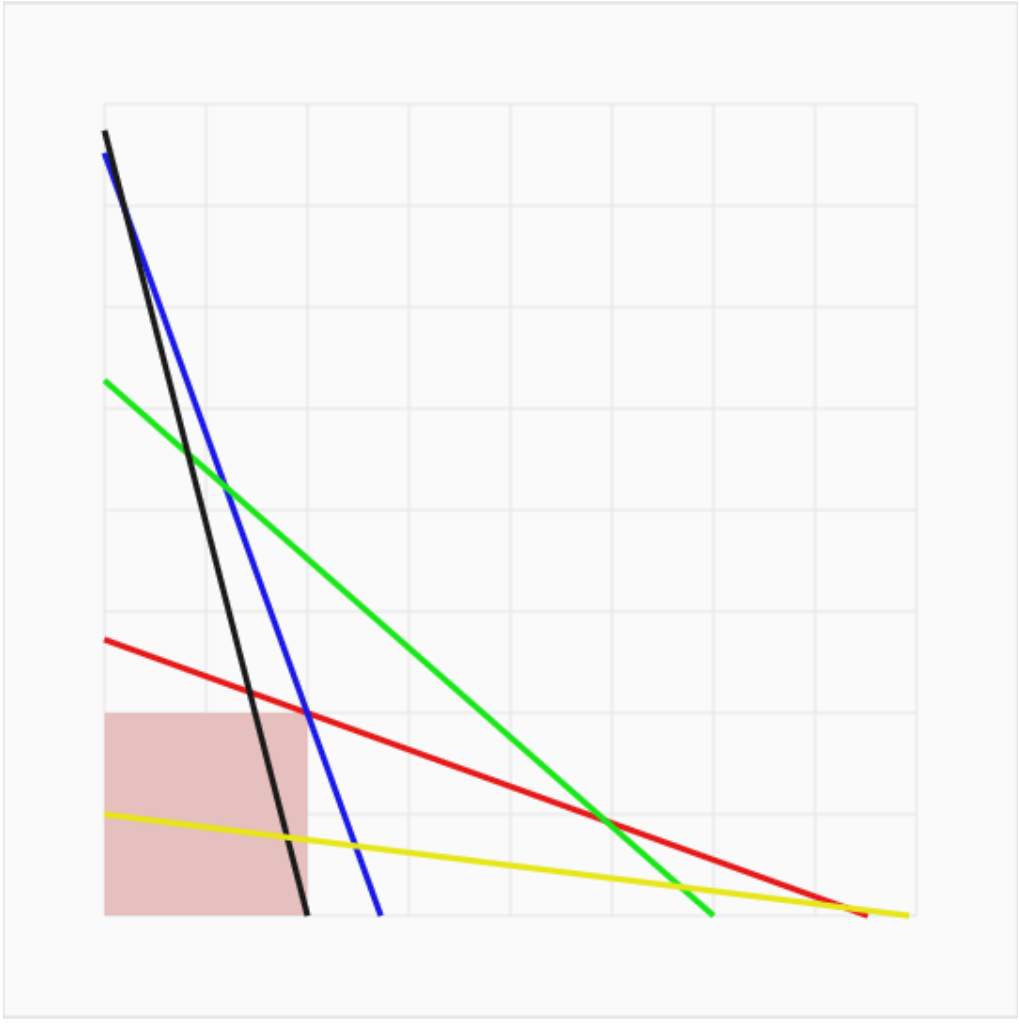


FUNCTION 2: The Satellite Function

```
double f2 ( double x )
{
  double value;

  value = 1.082 * cos ( x ) + sin ( x );

  return value;
}
```

FUNCTION 3: The Pole Equation

When does a 4 foot pole just touch the corner of a 1 foot square?



FUNCTION 3: The Pole Equation

Can a 4 foot pole leaning against a wall in such a way that it exactly touches the corner of a 1 foot cube?

The pole forms one big triangle against the wall and ground. Subtract the square from that triangle and two similar triangles remain, with horizontal sides x and 1, and with slanting sides y and $4 - y$. Because these triangles are similar:

$$\frac{y}{x} = \frac{4-y}{1}$$

Using the Pythagorean theorem, we can conclude that we need to find a value x for which it is true that:

$$x^4 + 2x^3 - 14x^2 + 2x + 1 = 0$$

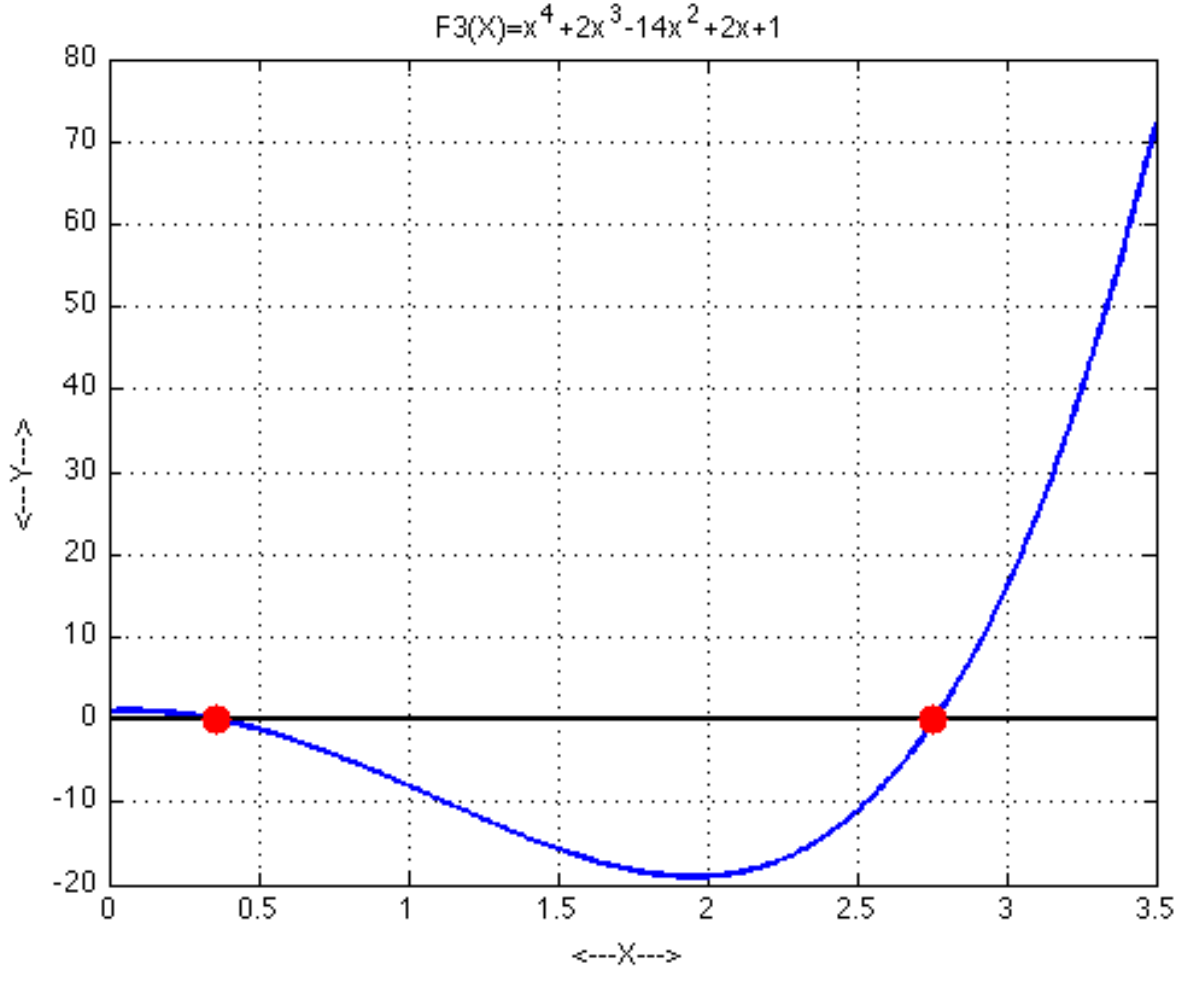
If we know x, we can determine y and the angle of the pole.

FUNCTION 3: The Pole Function

```
double f3 ( double x )
{
  double value;

  value = pow ( x, 4 ) + 2 * pow ( x, 3 )
    - 14 * pow ( x, 2 ) + 2 * x + 1;

  return value;
}
```

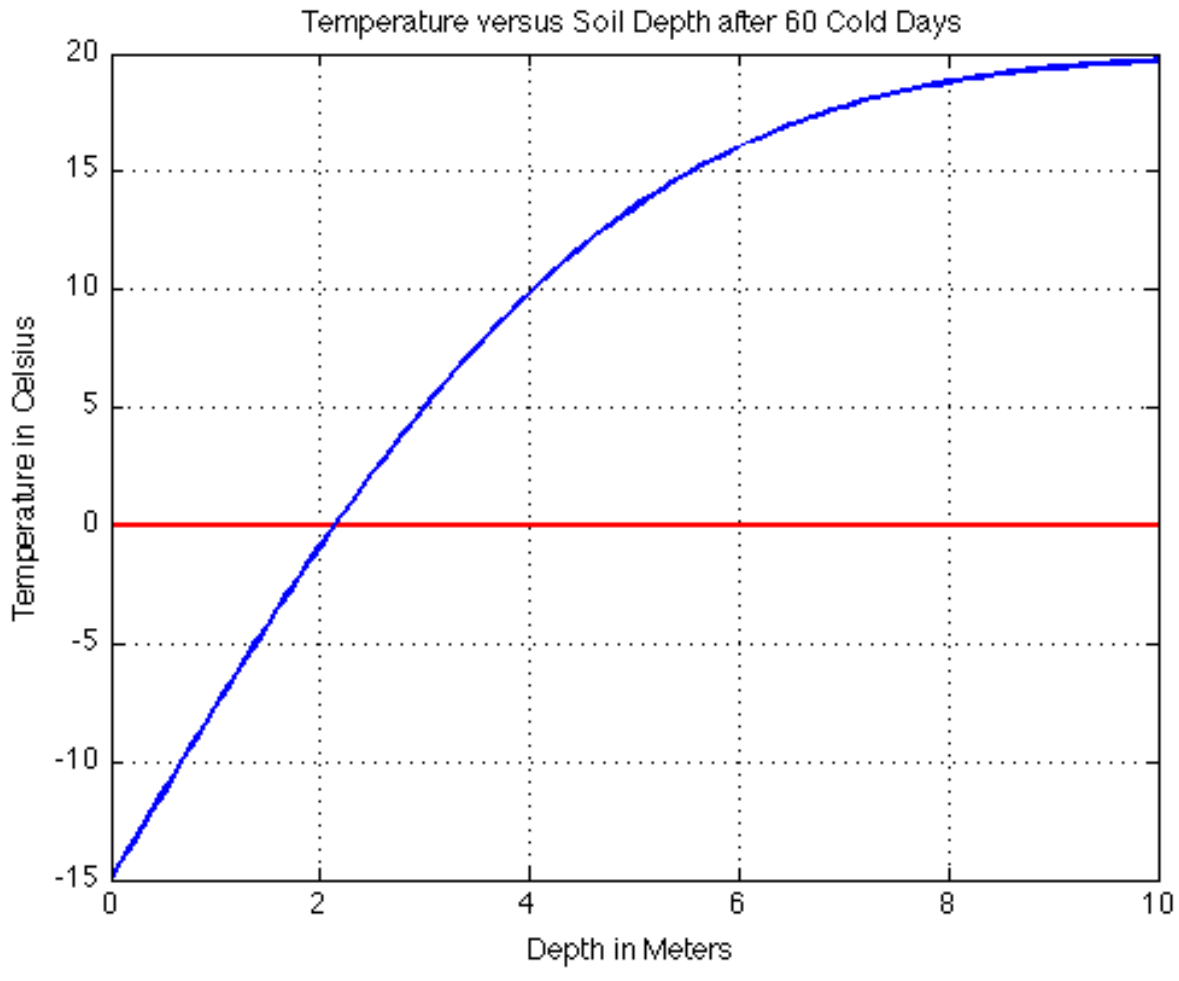
FUNCTION 3: The Pole Equation

The blue line is our function; the two red dots are solutions!



FUNCTION 4: The Frozen Pipe Equation

How deep should we bury a water pipe so it can survive a 60 day cold snap?



FUNCTION 4: The Frozen Pipe Equation

Buried water pipes will freeze if the surrounding soil cools down to 0 degrees Celsius. The deeper a pipe is buried, the longer it takes for cold weather on the surface to affect the pipe. In a simple model, we assume that initially, the soil had a temperature of T_i at all depths, and that a sudden cold snap means the surface air stays at a temperature of T_s.

$$\frac{T(x,t) - T_s}{T_i - T_s} = \text{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right)$$

where **erf()** is the *error function* and $\alpha = 0.138 \cdot 10^{-6}$ is a number that measures how rapidly the soil cools. The C++ math library includes the error function **erf()**. Time t is in seconds.

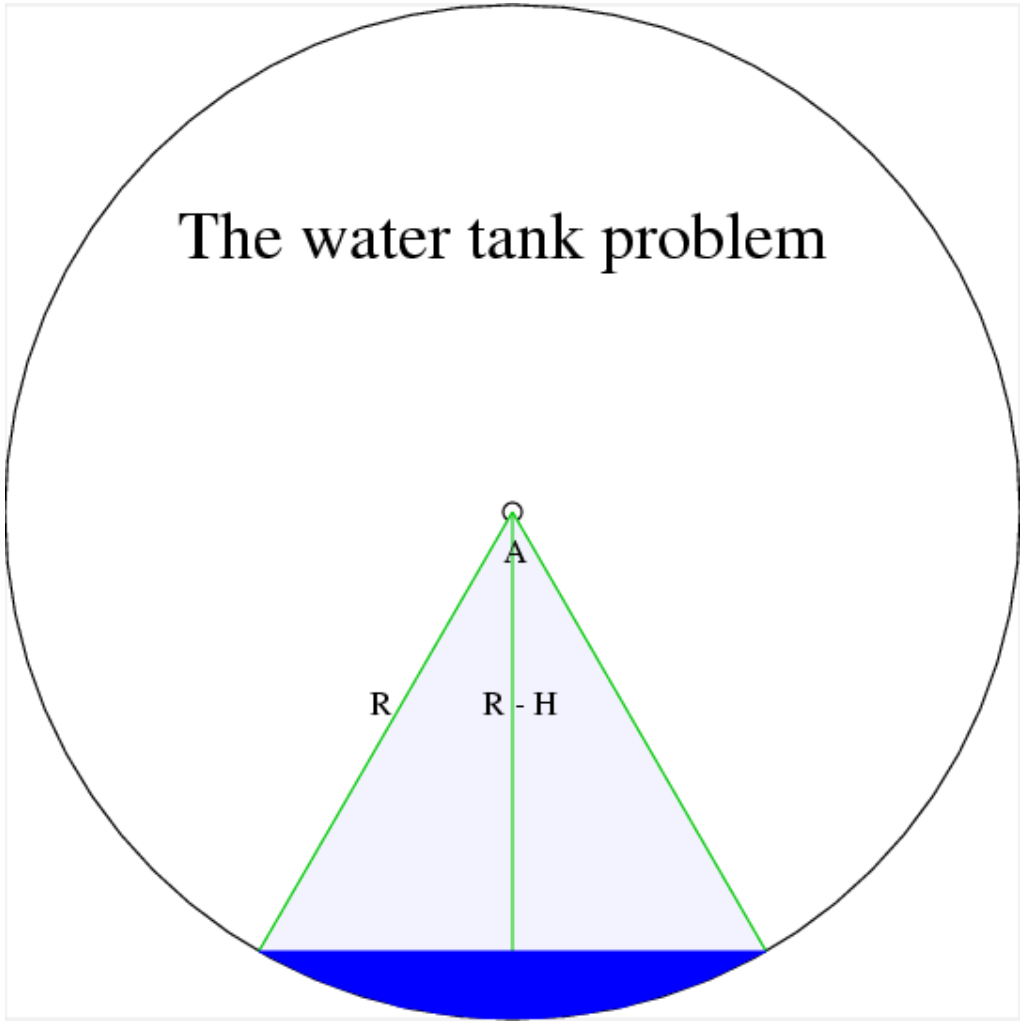
We seek a value x for which $T(x,t)$ is 0 when t equals 60 days (that is, the soil just got to freezing after 60 days).

FUNCTION 4: The Frozen Pipe Function

```
double f4 ( double x )
{
  double alpha = 0.138E-06;          <-- 0.138 x 10^(-6)
  double t = 60 * 24 * 60 * 60;      <-- days * hours *
                                         minutes * seconds

  double temp_init = 20.0;
  double temp_cold = -15.0;
  double value;

  value = temp_cold + ( temp_init - temp_cold )
    * erf ( 0.5 * x / sqrt ( alpha * t ) );

  return value;
}
```

FUNCTION 5: The Water Tank

A water tank has a circular cross section, and a radius of $R = 3$ feet. If the tank is $\frac{1}{4}$ full, what is the height H of the water?



FUNCTION 5: The Water Tank

Using formulas for the area of a circular sector, and of a triangle, we come up with the following formula for the area of the circle that contains water:

$$\begin{aligned}\text{Area} &= \text{circular sector} - \text{triangle} \\ &= R^2 \arccos\left(\frac{R-H}{R}\right) - (R-H)\sqrt{H(2R-H)}\end{aligned}$$

For our problem, we would be asking for a value of H that makes this equation true:

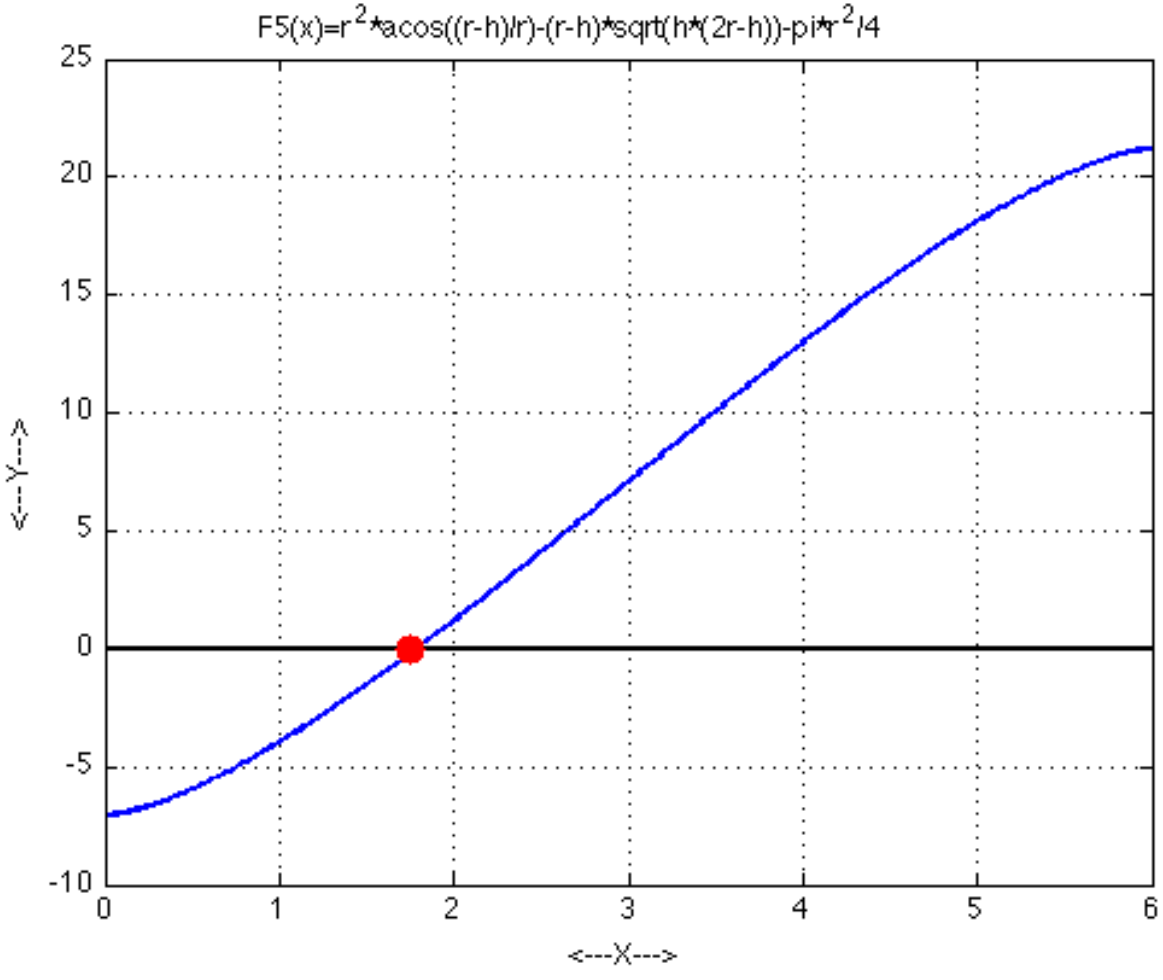
$$\frac{1}{4}(\pi 3^2) = 3^2 \arccos\left(\frac{3-H}{3}\right) - (3-H)\sqrt{H(6-H)}$$

which seems impossible to work out.

FUNCTION 5: The Water Tank Function

```
double f5 ( double h )
{
  double r = 3.0, pi = 3.14159265;
  double value;

  value = r * r * acos ( ( r - h ) / r )
    - ( r - h ) * sqrt ( h * ( 2 * r - h ) )
    - 0.25 * pi * r * r;

  return value;
}
```

FUNCTION 5: The Water Tank



FUNCTION 6: Lambert's Function

For our in-class exercise, we considered a special case of Lambert's function, by asking for a solution of the equation:

$$xe^x = 1000$$

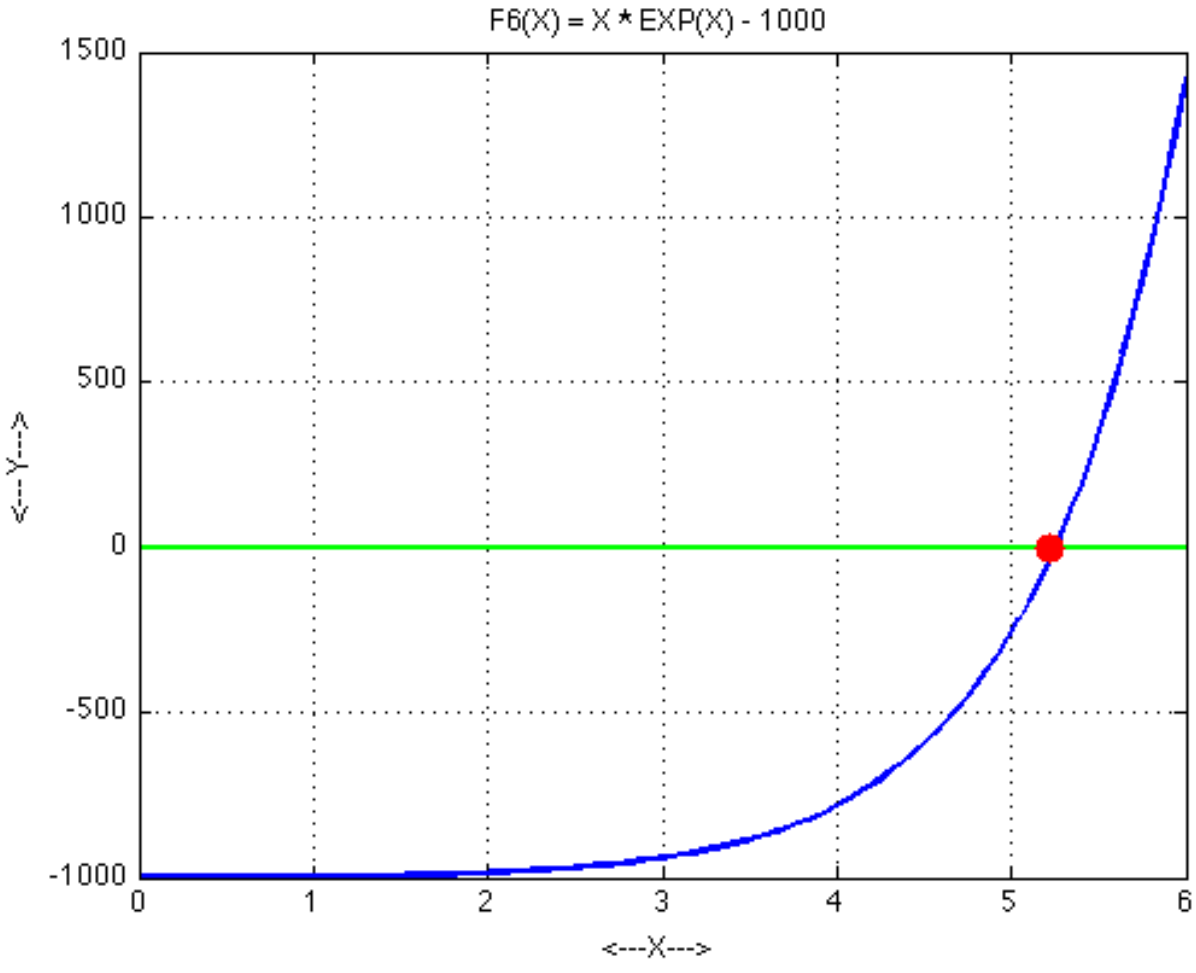
and we rewrote this as the function $f_6(x) = x * e^x - 1000$.

During the lab, you were probably able to find that the function changes sign in the interval [5,6].

FUNCTION 6: The Lambert Function

```
double f6 ( double h )
{
  double value;

  value = x * exp ( x ) - 1000.0;

  return value;
}
```

FUNCTION 6: The Lambert Function



Solving Better

- Introduction
- Making the Equation a Variable
- Sample Functions
- **The Secant Method**
- Newton's Method
- Assignment #7

SECANT: Can We Go Faster?

The bisection method has been good to us; it requires a change of sign interval, but after that, it slowly but surely narrows in on the solution. It takes 10 steps to reduce the size of the x interval by a factor of 1000, and maybe 20 steps to reduce it by a factor of 1,000,000. So, roughly speaking, if we started in the interval [0,1], it takes 20 steps to get 6 digits of accuracy in the solution.

If the slow but reliable bisection method is not good enough, you can try a quicker but less reliable procedure called the secant method. The secant method does not require a change of sign interval; its convergence can be significantly faster than bisection; however, it is not guaranteed to converge, especially if your starting estimate of the solution is too far from the correct v lu

SECANT: A Sequence of Pairs of Points

The secant method is an iteration that produces a sequence of estimates for the solution. It starts with estimates x_0 and x_1 for the solution, and produces x_2, a better estimate. We discard x_0 and use x_1 and x_2 for the next step, and so on.

The secant method approximates the the graph of the function by the straight line through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. If the curve and line are close, then we can hope that where the line crosses the x axis is close to where the curve crosses.

Since we only have two points of the curve at any one time, you can see we are taking quite a risk. However, if our procedure gets close to the solution, then the approximation should be good and get better very fast.

SECANT: Estimate the Crossing

If we have a line that goes through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$, then the formula for this line is

$$y = f(x_0) + (f(x_1) - f(x_0)) * \frac{x - x_0}{x_1 - x_0}$$

To find where this line crosses the x axis, we set y to 0 and solve for x, which we call x_2 because it's the next entry in our sequence:

$$x_2 = \frac{f(x_1) * x_0 - f(x_0) * x_1}{f(x_1) - f(x_0)}$$

We repeat the step, replacing x_0 by x_1, and x_1 by x_2, as needed.

Bisection always looks halfway between the two values. The secant method uses the size of the functions for a better guess.

SECANT: Code for the Method

secant.cpp

```
double secant ( double x0, double x1, double f ( double x ) )
{
  double fx0, fx1, fx2, x2;

  fx0 = f ( x0 );                                    <-- Initialization.
  fx1 = f ( x1 );

  while ( true )
  {
    x2 = ( fx1 * x0 - fx0 * x1 ) / ( fx1 - fx0 ); <--- Next estimate.
    fx2 = f ( x2 );

    if ( fabs ( fx2 ) < 0.000001 )                   <--- Can we stop?
    {
      break;
    }

    x0 = x1;                                         <--- Shift data.
    fx0 = fx1;
    x1 = x2;
    fx1 = fx2;
  }

  return x2;
}
```

SECANT: Calling the Method

secant_f1.cpp

```
# include <cstdlib>
# include <iostream>
# include <cmath>

using namespace std;

double secant ( double a, double b, double f ( double x ) );
double f1 ( double x );

int main ( )
//
//  SECANT_F1 is a program which uses SECANT to solve for a solution
//  of F1(X) = 0.
//
{
  double a = -10.0, b = +10.0, c;

  c = secant ( a, b, f1 );

  cout << "\n";
  cout << "SECANT returned solution estimate C = " << c << "\n";
  cout << "F1(C) = " << f1 ( c ) << "\n";

  return 0;
}
```

...text of secant()...
...text of f1()

SECANT: Solving COS(X) - X = 0

If we start the secant iteration with the same points we used for the bisection method, then here is the sequence of steps:

STEP	X	F(X)
0	-10.0	9.16093
0	10.0	-10.8391
1	-0.839072	1.50723
2	0.484153	0.400916
3	0.963678	-0.393174
4	0.726253	0.021415
5	0.738517	0.000951081
6	0.739087	-0.00000271414
7	0.739085	0.000000000340712

As the approximation gets close, the error really starts to drop. In fact, if we take one more step, the function value decreases by another factor of 10,000. In contrast, the bisection method to 23 steps, and did not get faster as we came closer.

SECANT: Starting without a Change of Sign

Note that the secant method does not actually need a change of sign interval, either. We can start with the points x0 = - 10.0 and x1 = -8.0. Here's what we get then:

STEP	X	F(X)
0	-10.0	9.16093
0	-8.0	7.85450
1	4.02439	-4.65938
2	-0.452739	1.35199
3	0.554191	0.296135
4	0.836604	-0.166616
5	0.734920	0.00696489
6	0.739000	0.000142883
7	0.739085	-0.000000131713

SECANT: Defeating the Secant Method

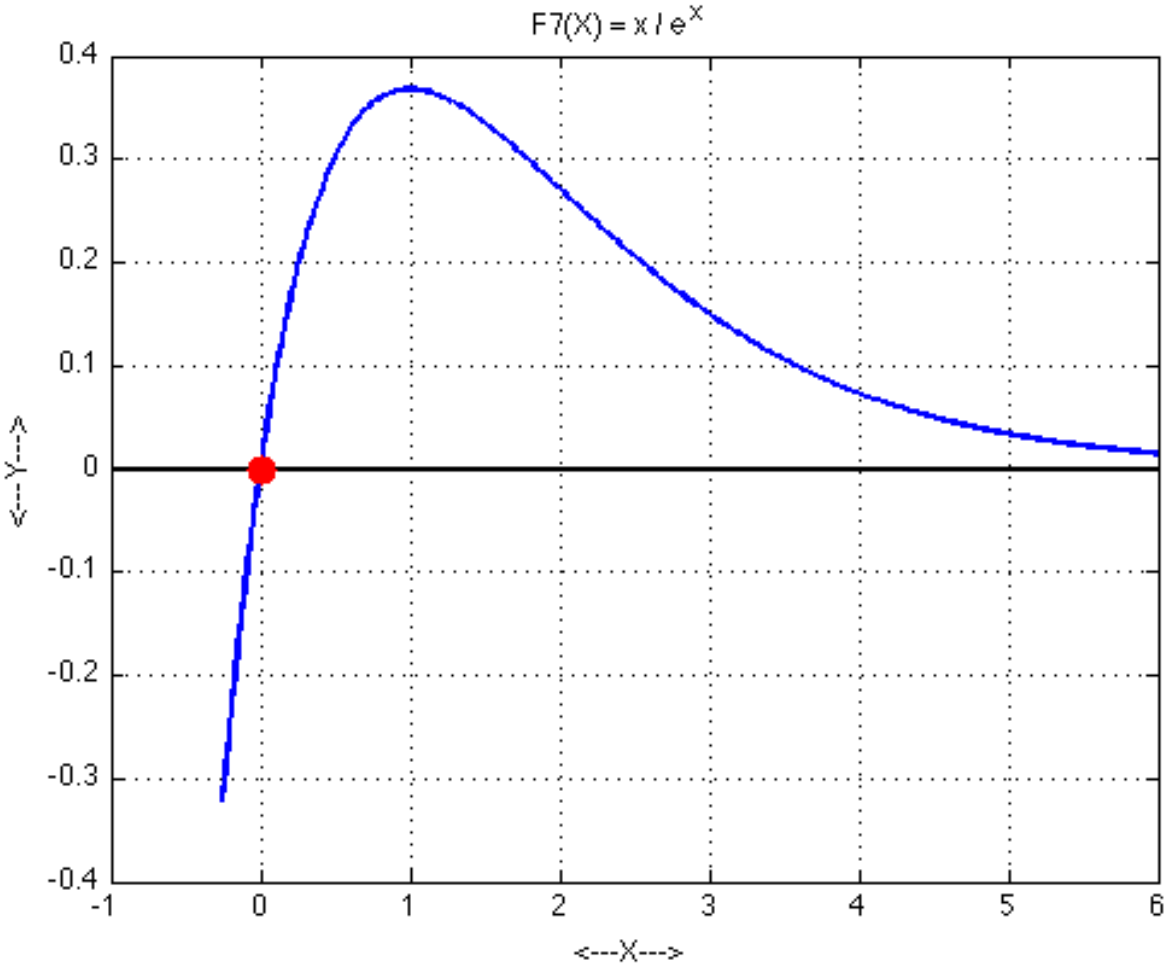
However, the secant method works best when the curve is not too wiggly. To defeat it, we need a situation where the graph is going down, but **not** towards a solution!

Here is a table of data for such a case. The solution is at $x = 0$, but the secant method is running away from the solution. Why?

STEP	X	F(X)
0	1.5	0.334695
0	2	0.270671
1	4.1138	0.0672424
2	4.81251	0.0391135
3	5.78407	0.0177928
4	6.59487	0.00901761
5	7.42806	0.00441477
6	8.22722	0.00219896
7	9.02029	0.00109083
8	9.80099	0.000542944
9	10.5746	0.000270245
10	11.3413	0.000134644

SECANT

Going down the hill on the right guarantees you'll never find a solution!



The Secant and Newton Methods

- Introduction
- Making the Equation a Variable
- Sample Functions
- The Secant Method
- **Newton's Method**
- Assignment #7

NEWTON: Rewrite the Secant Method

The secant method works (when it works) by using the *current position* and an estimate of the slope of the graph.

We can rewrite the secant formula as follows:

$$\begin{aligned} x_2 =& \frac{f(x_1) * x_0 - f(x_0) * x_1}{f(x_1) - f(x_0)} \\ =& x_1 - f(x_1) * \frac{x_1 - x_0}{f(x_1) - f(x_0)} \\ \approx& x_1 - \frac{f(x_1)}{f'(x_1)} \end{aligned}$$

because

$$f'(x_1) \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

NEWTON: Use Derivative instead of Slope Estimate

Notice that in this new version, the point x_0 has completely disappeared. It seems we really only needed it to estimate the slope of the curve; if we have the derivative at x_1, that gives us the information we need.

This suggests two important advantages of Newton's method:

- it only needs one point at a time;
- the derivative is a more accurate estimate of the slope, which means the computation might be faster.

and, of course, a disadvantage:

- the user must supply a derivative function.

NEWTON: The Derivative Function (Careful!)

The function **fp1()** should evaluate the derivative of $f1(x) = \cos(x) - x$.

```
double fp1 ( double x )
{
  double value;

  value = sin ( x ) - 1;

  return value;
}
```

We will need to come back and fix this function shortly!

NEWTON: The Newton Method Fails

Let's start at $x_0 = -10$ and see how Newton's method works:

STEP	X	F(X)	
0	-10.0000	9.16093	
1	10.0907	-10.877	
2	3.36722	-4.34187	
3	-0.180882	1.16457	
4	0.806126	-0.113827	
5	0.39725	0.524878	
6	1.25333	-0.941176	
7	-17.5816	17.8799	
8	375.219	-375.419	
9	185.588	-186.561	
10	34.1486	-35.0662	
11	-24.0607	24.539	
12	177.367	-177.235	
13	-19963.9	19963.3	
14	-8925.65	8924.72	
15	5397.85	-5397.03	<-- 15 step limit stops us!

NEWTON: Oops

So after looking at these discouraging results, let's go back and look at the function and derivative codes that we gave **newton()** to work with:

```
f1():     value = cos ( x ) - x;
fp1():    value = sin ( x ) - 1;    <-- Notice anything wr
```

If we give **newton()** wrong information, we can't blame it for getting confused. But we have to realize that when the user has to supply **two** pieces of information, the function and the derivative, we've doubled the chances that the user (in this case, me!) is going to make a mistake.

Let's rerun the program after fixing my error!

NEWTON: The Newton Method Fails Again

Let's restart, using the correct derivative!

```
STEP         X          F(X)
----  -------------   --------
   0        -5            5.28366
   1      -2.30277        1.63443
   2       4.0781        -4.6707
   3      -19.9346       20.4015
   4      156.443       -155.639
   5     -227.134        227.724
   6      953.782       -953.478
   7   -19172.3         19171.6
   8    58810.8        -58809.8
   9    10096.6        -10095.7
  10    -8064.42         8063.42
  11      380.285       -381.273
  12      -68.9853        69.9769
  13       -7.02201        7.76127
  14       16.7432       -17.2535
  15     -106.486        107.433  <-- 15 step limit stops us!
```

These results are just as bad, and we're using good information.
Our starting point must be too far away!

NEWTON: Starting Close, the Method is Fast

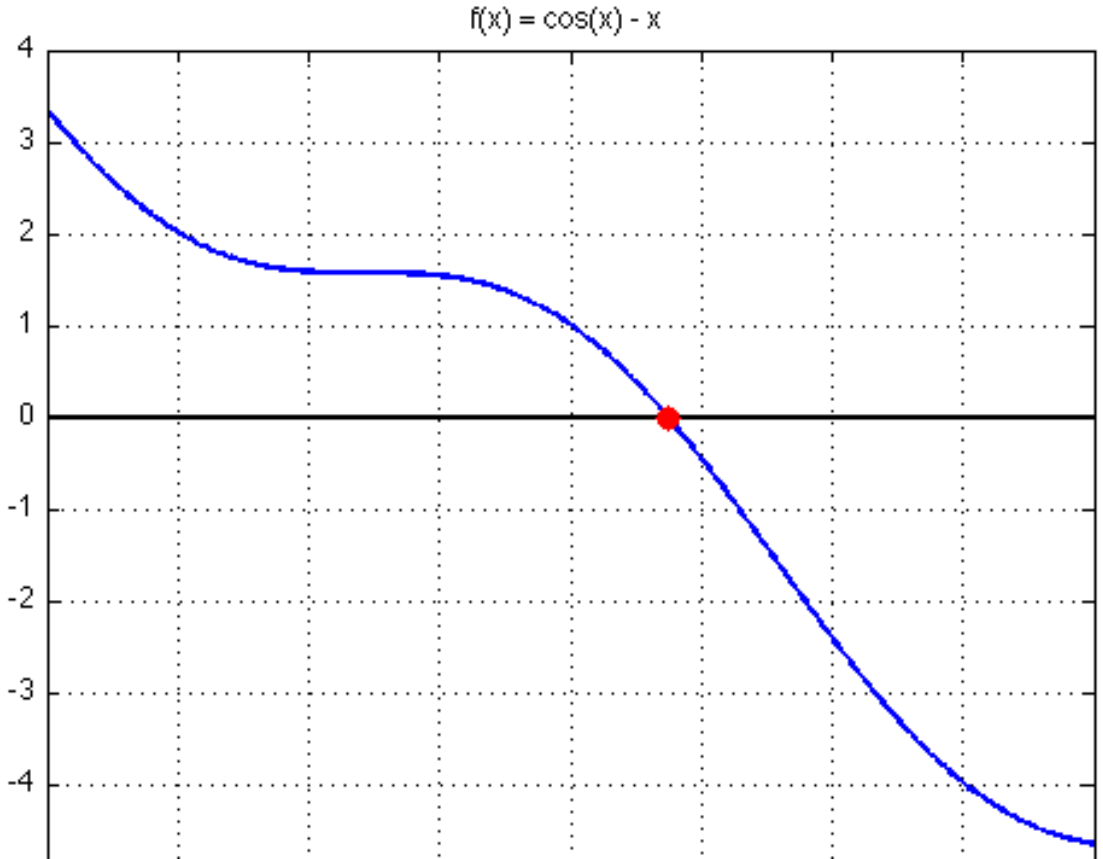
Let's start at 0, which is one endpoint of the change of sign interval we used with bisection (which took 23 steps to get a decent result.)

```
STEP      X          F(X)
----  ----------   --------
   0    0            1
   1    1           -0.459698
   2    0.750364    -0.0189231
   3    0.739113    -4.64559e-05
   4    0.739085    -2.8472e-10
```

Once we get close enough that the graph is behaving like a straight line, **newton()** picks up the answer, and the error drops very quickly.

NEWTON: The Cosine Equation Has Flat Spots

When our function $f_1(x)$ is roughly flat, such as between $-2.5 \leq x \leq -1$, the derivative is almost zero, and suggests going way far to the right for the next approximation. It is only if we are in the range $-0.5 \leq x \leq +3$ that I feel "safe" that that **newton()** can figure things out.



The Secant and Newton Methods

- Introduction
- Making the Equation a Variable
- Sample Functions
- The Secant Method
- Newton's Method
- **Assignment #7**

ASSIGNMENT #7: F3(X), the "Pole" Function

Consider the pole function,

$$f_3(x) = x^4 + 2x^3 - 14x^2 + 2x + 1$$

Because the equation $f_3(x) = 0$ involves a polynomial of degree 4, it is possible to have as many as four solutions, and in this case, we have exactly that many. For the pole problem, the two positive solutions are most interesting, but there are also two negative solutions.

ASSIGNMENT #7: Solve the Pole Equation

- Using plotting, or the **bracket** function, try to find change of sign intervals for the four solutions to the pole function problem. *(four pairs of numbers [a,b])*
- For each interval, use the secant method to estimate the solution. Start the secant method with the pair of endpoints of your interval. *(four solution estimates)*
- Compute the derivative function fp3(x), and use it, with Newton's method, starting at $x = 2$, to compute one solution to the problem. *(one solution estimate)*

ASSIGNMENT #7: Things to Turn in

Email to Detelina:

- your program's results:
  - 4 pairs of change of sign intervals,
  - 4 secant answers,
  - 1 Newton answer
- a copy of your programs.

The program and output are due by Thursday, July 14.