Intro Math Problem Solving
October 19

plotf example: a function name as input

Our Bisection Code

What Can Go Wrong?

Functions with multiple zeros

Faster solutions: secant method

Safer, faster solutions: false position

How MATLAB solves equations: fzero

Anonymous functions

Homework #7

Office Hours October 23/25/27

October 23: Monday, 2-3 only

October 25: Wednesday, 2-4, as usual

October 27: Friday, 2-2:30 only

Location is the same:

Monday/Wednesday in Torgerson 3050

Friday in Newman Library Second Floor

References

Chapter 9, Section 3 of our textbook discusses these topics, and can be useful for comparison and background to these notes.

"Insight Through Computing" is available as an ebook on the library web site, and chapter 9 is also in today's Canvas folder.

Cleve Moler, the "inventor" of MATLAB, has an e-book "Numerical Computing with MATLAB", which includes a chapter "Zeros and Roots", which discusses many of the topics we will look at today.

Chapters of that book are available at:

https://www.mathworks.com/moler/chapters.html

One function as input to another function

We wrote "function plotshape(x,y,color)" that can make a line plot, with given color, of a polygon whose vertices are listed in x and y.

Could "function plotf(a,b,f,color)" make a line plot of the function f(x), for a <= x <= b?

Yes, except that MATLAB regards a function "f" as a special kind of input. "f" is not a variable or a value, it's the name of a function. To indicate that it's a different kind of input, we have to precede its name with an AT sign, "@".

Function to Plot a Function

```
function plotf ( a, b, f, color )

  x = linspace ( a, b, 501 );
  y = f(x);          %  <- We assume the function's name is "f".
  plot ( x, y, 'Linewidth', 3, 'Color', color );
  grid on
  xlabel ( '<-- X -->' );
  ylabel ( '<-- Y -->' );
  title ('I don''t know what function this is!' );

  return
end
```

Using plotf.m

When calling plotf(), the actual input corresponding to "f" can be any function which has 1 input and 1 output:

```
a = 0.0;
b = 2 * pi;
color = [1.0, 0.4, 0.0 ];
plotf ( a, b, @sin, color );  <- MATLAB sin

plotf ( 0, 1, @cosxx, 'r' ); <- user function
```

Functions as Input

It's a little hard, at first, to get used to the idea that the name of a function can be used as input, just as we can pass numbers and variables.

For plotting, zero finding, or other computations, it's common to write a code that works for any function the user cares to name.

The "@" sign specifies the particular function you have in mind. Meanwhile, the function doing the work behaves as though it's working with a function named "f", or whatever temporary name is used.

```
          plotf ( 0.0, 1.0, @cosxx, 'r' );
                    |    |      |         |
function plotf ( a,   b,     f,         color );
```

We use this idea to make our bisection function flexible.

Avoid Confusion!

The ONLY time you need an @ sign is when you are marking the name of a function that is to be input to another function:

```
plotf ( 0.0, 1.0, @cosxx, 'r' );  <- Calling plotf, cosxx is a function!
```

You DON'T use an @ to evaluate the function!

```
y = cosxx(7)     <- YES
y = @cosxx(7)  <- You are SO wrong!
```

You DON'T use an @ in function headers:

```
function plotf ( a, b, f, color )      <- YES
function plotf ( a, b, @f, color )   <- This is NOT right!
```

RULE: an @ sign turns a function name into an input to another function.

A Simple Bisection Script

Last time, we thought about the problem of finding a value X for which a given function F returns a value of 0, that is, we want to solve the equation f(x)=0.

We decided to insist that we would only try to do this if F was continuous, and if we were given values A and B at which F had opposite signs.

Under these conditions, we invented the idea of bisection: compute C as the average of A and B, look at F(C), and have C replace whichever of A and B has the same function sign as F(C).

bisection1.m

```
function x = bisection1 ( xtol, a, b, f )
  while ( xtol < b - a )
    c = ( a + b ) / 2.0;
    if ( sign ( f(c) ) == sign ( f(a) ) )   <-  sign (*) = -1, 0, or +1.
      a = c;
    else
      b = c;
    end
  end
  x = ( a + b ) / 2.0;    <- Use midpoint of final [a,b] as answer.
  return
end
```

Good Things About Bisection

Because F is continuous, and [A,B] is a change-of-sign interval, we know there is a solution X within [A,B].

Each bisection step cuts the size B-A in half. Eventually, it must become smaller than XTOL.

Near the exact solution, F must get small in magnitude. Therefore, if F(C) gets very small, C is probably near the solution.

Probably Good Things About Bisection

Because F is continuous, and [A,B] is a change-of-sign interval, we know there is a solution X within [A,B]. (On a computer, is this absolutely true?)

Each bisection step cuts the size B-A in half. Eventually, it must become smaller than XTOL. (Is this absolutely true?)

Near the exact solution, F must get small in magnitude. Therefore, if F(C) gets small enough, C is probably near the solution. (Is that really true?)

|F(X)| < FTOL, not the best test

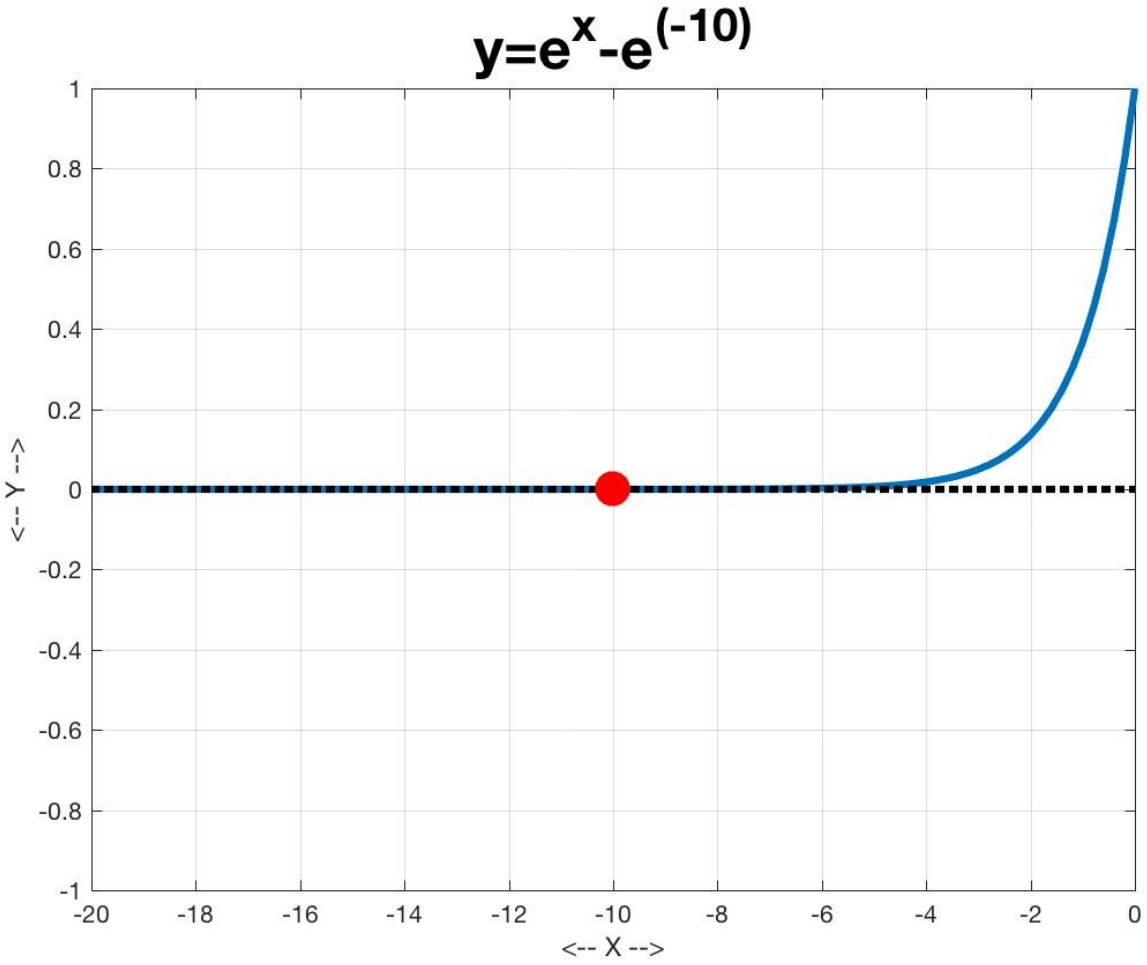
We will hope to find intervals [A,B] smaller than XTOL, containing a solution X.

We might also be interested in values X for which |F(X)| < FTOL, but this is a less reliable test.

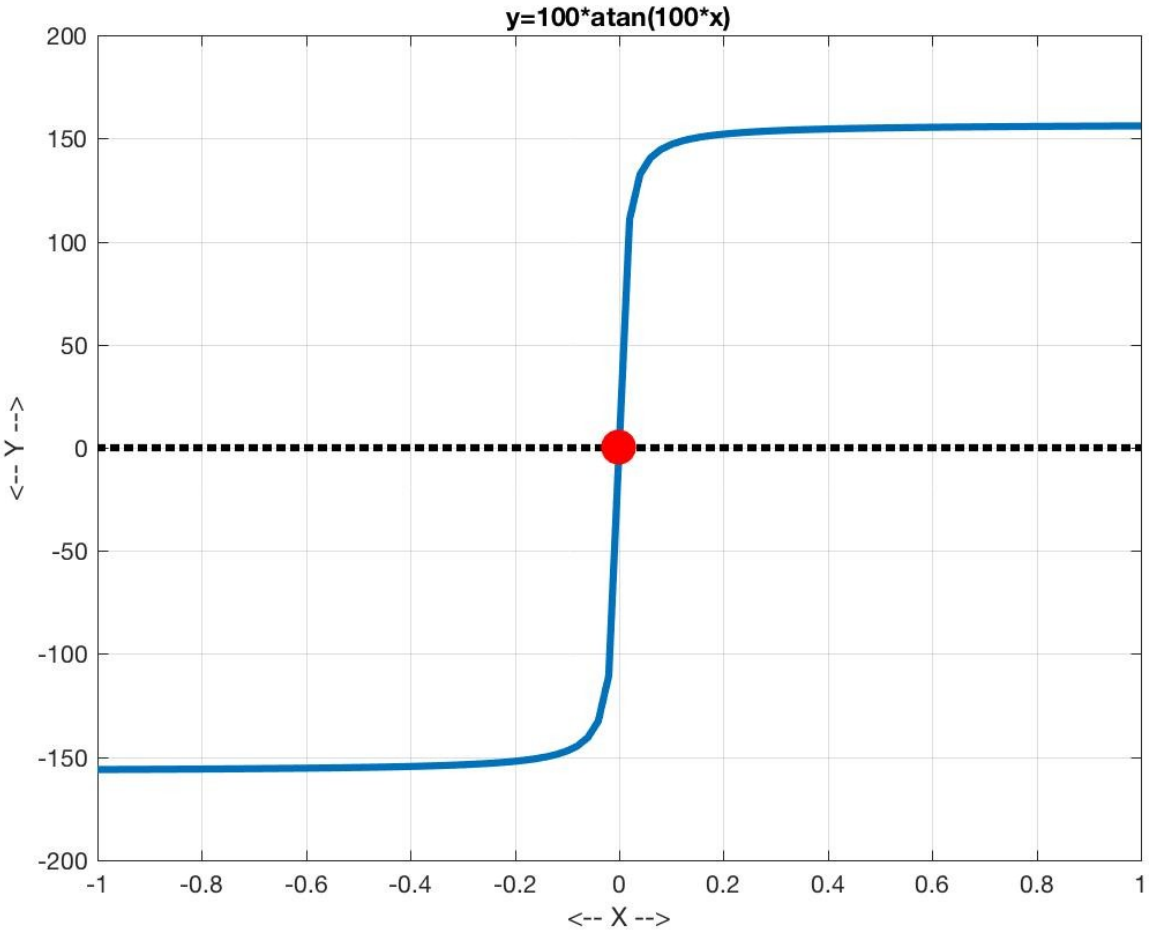
One bad case occurs if f(x) is very small over a very wide range, such as e^x for negative x: too many approximate answers.

Another bad case occurs if the function has very large magnitudes near the zero: not enough (very hard to find) approximate answers.

|f(x)| < FTOL, interval very wide



|f(x)| < FTOL, interval very narrow



bisection2.m

```
function [ x, a, b ] = bisection2 ( xtol, a, b, f )    <- Return new values of A and B.

  if ( sign ( f(a) ) * sign ( f(b) ) ~= -1 )              <- Check for change of sign.
    error ( 'F(A) and F(B) are not of opposite signs!' );
  end

  while ( xtol < b - a )

    c = ( a + b ) / 2.0;

    if ( sign ( f(c) ) == sign ( f(a) ) )
      a = c;
    else
      b = c;
    end

  end

  x = ( a + b ) / 2.0;

  return
end
```

Using bisection2

Recall our question about the Theron formula for the weight of a child:
"At what age do you predict my child will weigh 1000 pounds?"

Recall our Theron formula:

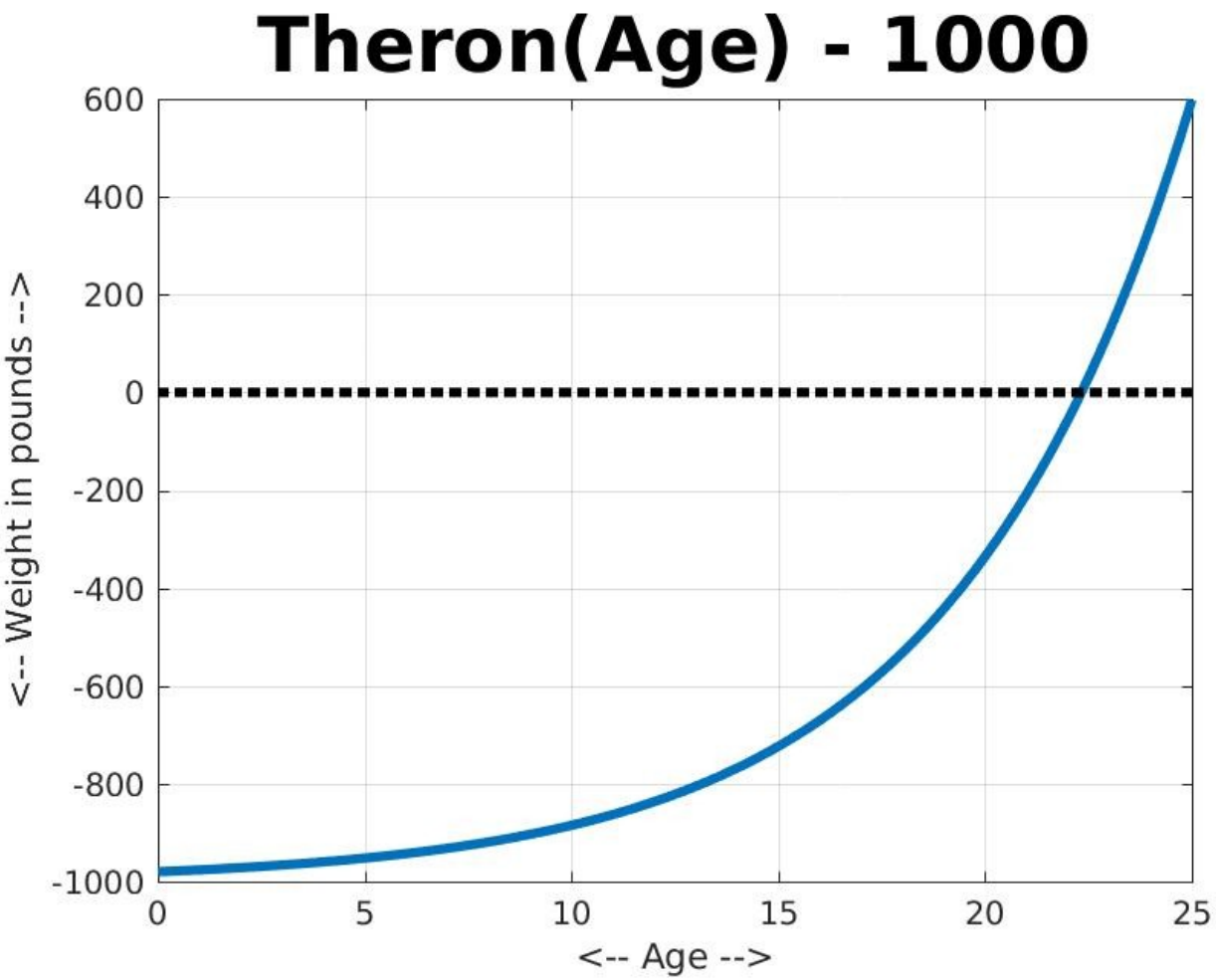
```
function weight = theron ( age )
  weight = 2.20462 * exp ( 0.175571 * age + 2.197099 );
  return
end
```

We're asking to solve
  theron(age) = 1000
To use our bisection code, we need a new function:
  theron_1000(age) = theron(age) – 1000
so that bisection2 can try to solve:
  theron_1000(age) = 0

Our Theron_1000.m Code

```
function value = theron_1000 ( age )
  value = theron ( age ) - 1000.0;
  return
end
```

Get a Change of Sign Interval by Plotting



Solve with Bisection2

```
xtol = 0.00001;
a = 15.0;
b = 25.0;
[x, a, b ] = bisection2 ( xtol, a, b, @theron_1000 );

F(   22.32772350311279)        =  -0.000377371
Theron(   22.32772350311279)=   999.9996226293672
B-A = 9.5367e-06

A=   22.32771873474121       <    X =22.32772350311279     <    22.32772827148438        = B
FA=-0.001214557734328992 <   FX =-0.000377370632804741 < 0.0004598171680072483 = FB
TA=   999.9987854422657    <    TX= 999.9996226293672    <    1000.000459817168        = TB
```

We did NOT find an exact solution X.
But our answer X is inside an interval B-A of width less than XTOL, so X is closer than XTOL to the exact solution.

F(X) is NOT zero, but a weight of 999.9996 is...pretty close to 1000!

If our answers are not close enough, we can try again with a smaller XTOL.

The Mountain Climbing Problem

Bisection expects to solve f(x) = 0. To solve our Theron problem, we had to make a new function theron_1000.m, to satisfy this requirement.

The mountain climbing problem asks when the ascenders and descenders meet, and we have functions ascent.m and descent.m already written.

What we want is the time H so that

ascent(h) = descent(h);

To use bisection, we have to rewrite this as a function:

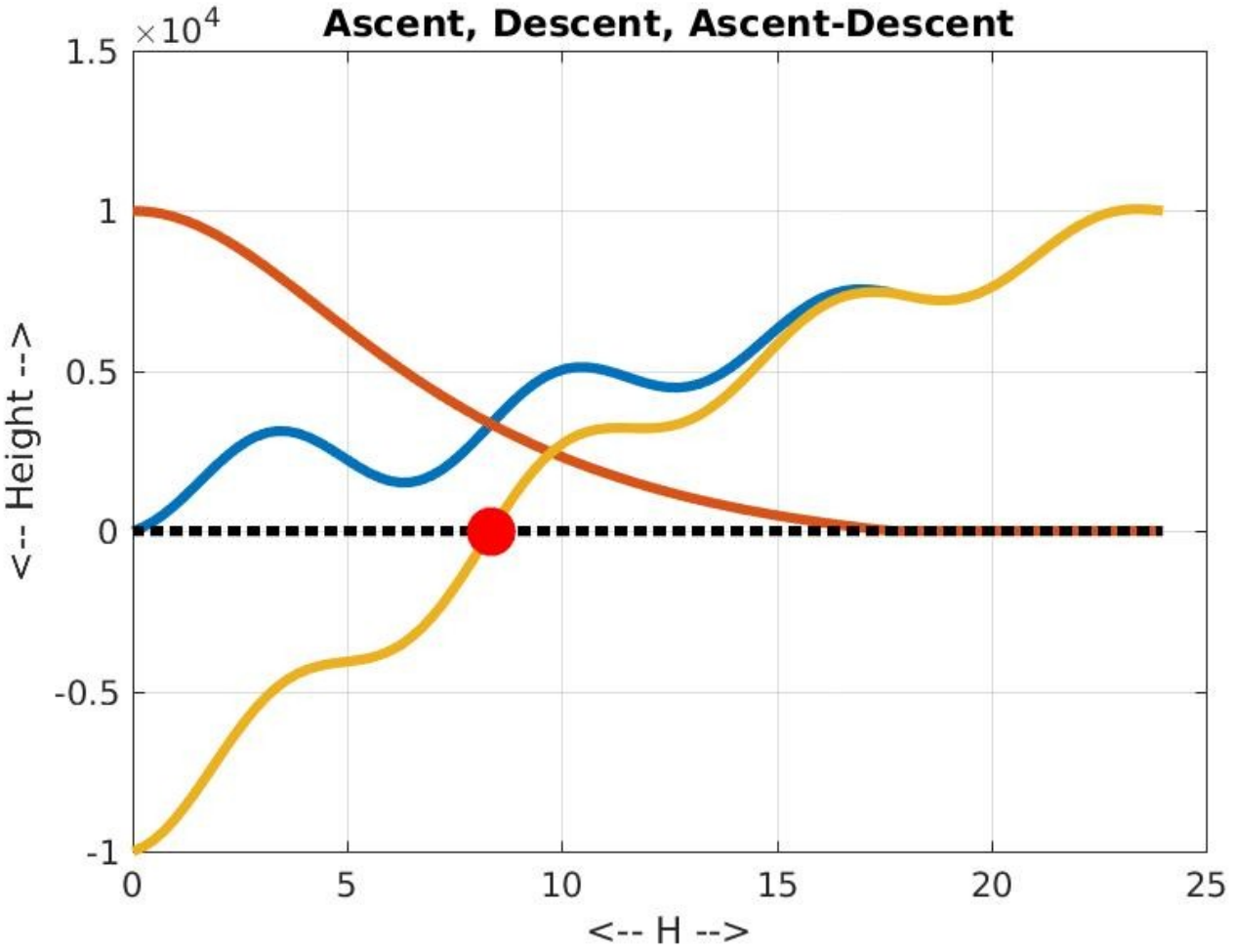
f(h) = ascent(h) – descent(h)

and now bisection can tell us when f(h) = 0.

Our new function looks like this:

```
function y = ascent_minus_descent ( h )
  y = ascent ( h ) - descent ( h );
  return
end
```

The Mountain Climbing Problem



Solve with Bisection2

```
xtol = 0.00001;
a = 0.0;
b = 24.0;
[x, a, b ] = bisection2 ( xtol, a, b, @ascent_minus_descent );
```

X = 8.3436... <- They met at about 8:20 AM

F(8.3436) = 0.0013 <- Might seem a little large...
but look at Ascent and Descent values!

Ascent(X) = 3346.799423540228

Descent(X) = 3346.798080463217

B-A = 5.7e-06

Try f(x) = polynomial

Let's try bisection on a new function:

tau(x) = 512 x^10 – 1280 x^8 + 1120 x^6 – 400 x^4 + 50 x^2 – 1

This polynomial of 10^{th} degree can have as many as 10 values x for which tau(x) = 0.

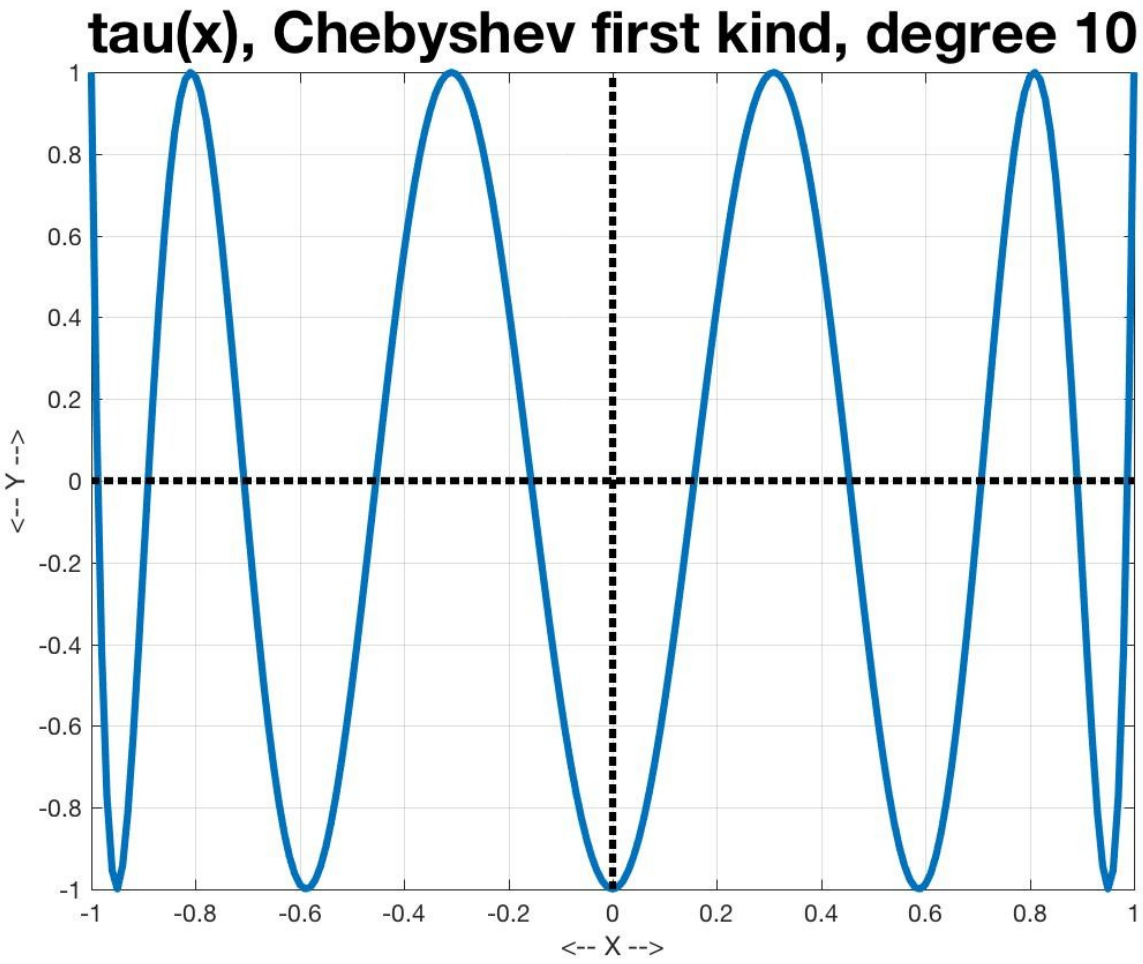
Given this choice, we will look for the smallest positive x.

A plot can help us see what is going on.

.

.

We seek the solution near x=0.15



Bisection2 on tau(x)

```
xtol = 1.0e-5
a = 0.0;
b = 0.3;
[ x, a, b ] = bisection2 ( xtol,  a, b, @tau );

Estimated zero F(  0.1564315795898437 ) = -2.92142e-05
Interval [   0.156427001953125,  0.156436157226562 5]
Interval width: 9.1553e-06
```

.

Computations have LIMITED accuracy

Our bisection algorithm accepts any tolerance. We know that MATLAB's arithmetic is really only accurate to about 16 digits. What happens if we ask for more accuracy than MATLAB can deliver?

It's not pretty, and it's another error without an error message! We will have to fix this!

Decrease XTOL Too Far!

xtol	x	width
1.0e-05	0.1564315795898 44	9.1e-06
1.0e-10	0.1564344650250 86	6.9e-11
1.0e-15	0.1564344650402 31	5.2e-16
1.0e-16	0.1564344650402 31	8.3e-17
1.0e-17	---program runs forever!---	

Different Number Systems

We are used to a mathematical model of the real numbers. In particular, between any distinct real numbers A and B, there infinitely many (uncountable, in fact) more values. In particular, (A+B)/2 is between A and B, and different from both of them.

In the computational model of the real numbers, if the distinct values A and B are sufficiently close:

* there are no more numbers between them!
* the value of (A+B)/2 will be either A or B "exactly"!

bisection3.m

```
function [ x, a, b ] = bisection3 ( xtol, a, b, f )

  bisect_max = 50;  bisect_num = 0;

  while ( xtol < b - a )

    bisect_num = biosect_num + 1;
   if ( bisect_max < bisect_num )
      fprintf ( 'Maximum number of steps exceeded!\n' );
      break
   end

    c = ( a + b ) / 2.0;
    if ( sign ( f(c) ) == sign ( f(a) ) )
      a = c;
    else
      b = c;
    end

  end

  x = ( a + b ) / 2.0;
  return
end
```

Run the tiny tolerance problem again

```
>> [x,a,b] = bisection3 ( 1.0e-17, 0.0, 0.3, @tau )

Maximum number of steps exceeded!
Bisection terminated without satisfying tolerance

Estimated zero F(0.156434465040231) = -2.2e-16
Interval [   0.156434465040231,  0.156434465040231]
Interval width: 8.3e-17
.
[A,B] appear to be the same value, but are not.
They just print as the same value...
```

bisection3.m warns us!

xtol	x	width
1.0e-05	0.156431579589844	9.1e-06
1.0e-10	0.156434465025086	6.9e-11
1.0e-15	0.156434465040231	5.2e-16
1.0e-16	0.156434465040231	8.3e-17
1.0e-17	0.156434465040231	8.3e-17
1.0e-18	0.156434465040231	8.3e-17
0.0	0.156434465040231	8.3e-17

Tolerance not achieved, but results useful.

Can we find ALL the zeros?

A linear equation, such as 4x+2=10, is easy to solve, and there is only one value x that makes it true.

But we already know that a polynomial equation of degree n might have as many as n distinct values x that make it true, called "zeros" or "roots".

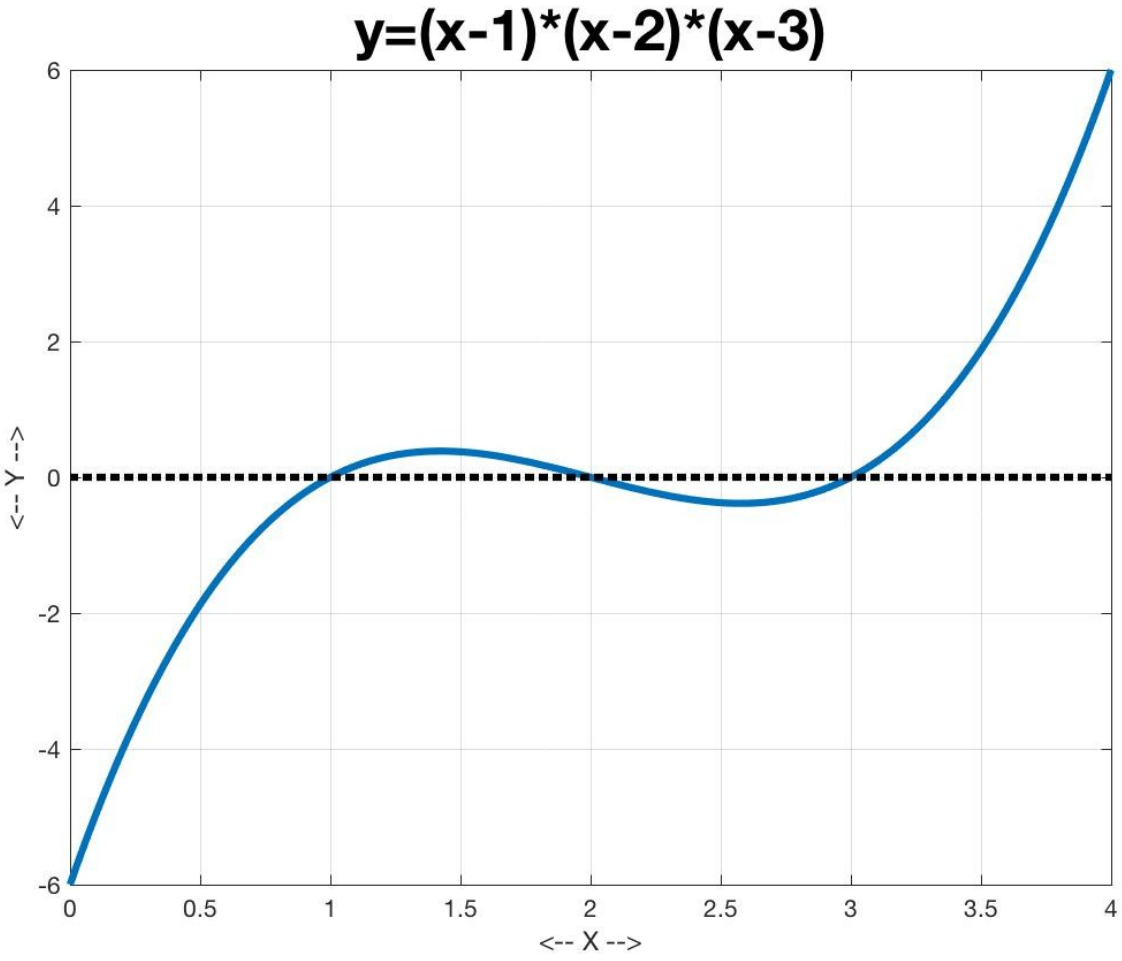
It's easy to construct an example:

f(x) = (x-1)*(x-2)*(x-3) = x^3-6x^2+11x-6

has roots x=1, x=2 and x=3.

Can bisection handle such a problem, and find all three roots?

The plot suggests where to look



"Fences" around each root

Bisection needs a change of sign interval to work.

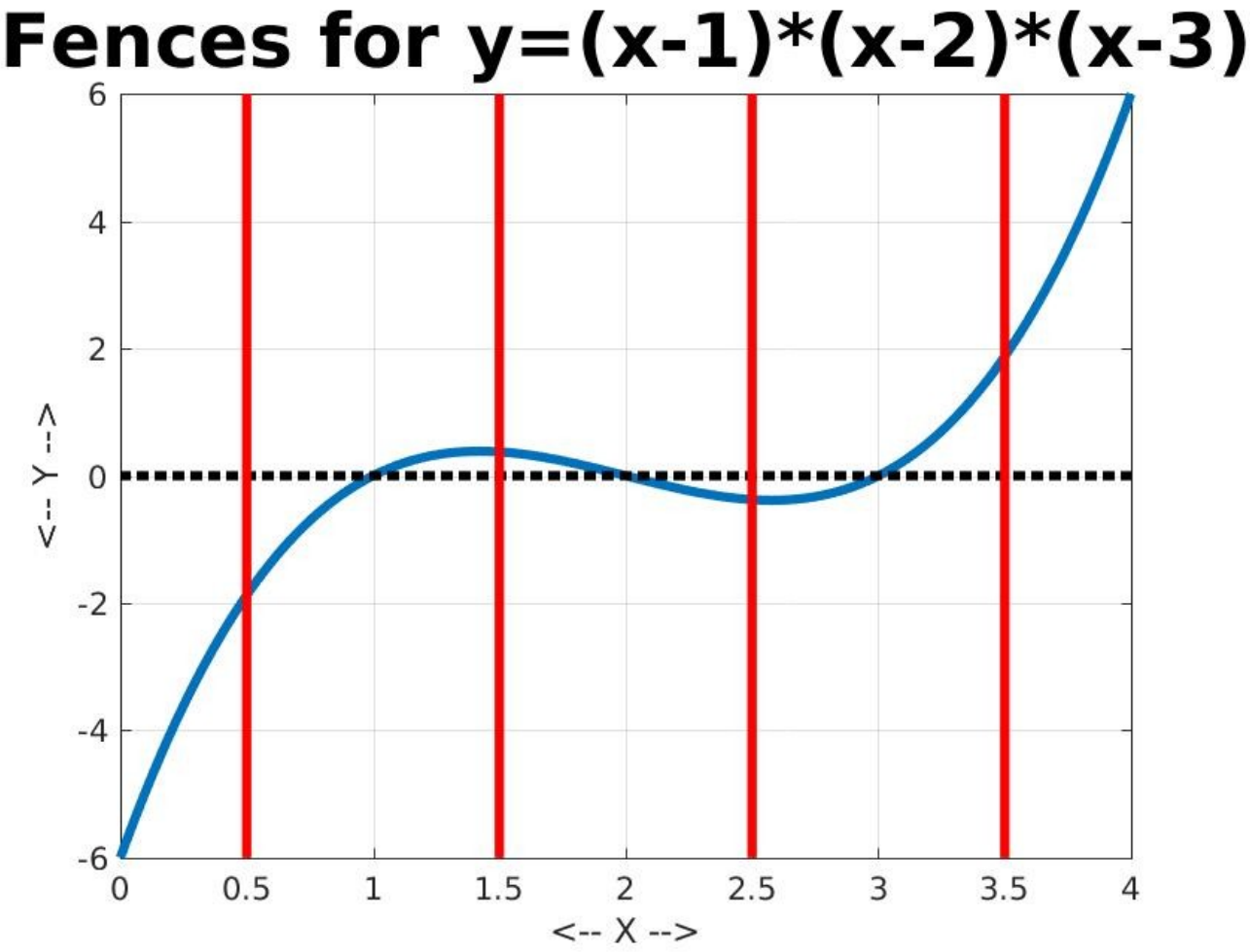
Even if we didn't know the formula for the function, we can see that it is negative around 0.5, positive around 1.5, negative again at 2.5 and positive around 3.5.

Think of 0.5, 1.5, 2.5 and 3.5 as "fence posts". Each pair of values defines a change of sign interval, and so inside each fence is a root somewhere.

---0.5---?---1.5---?---2.5---?---3.5---

To find the roots, we use bisection three times, once inside each "fence".

Do Bisection Inside Each Fence



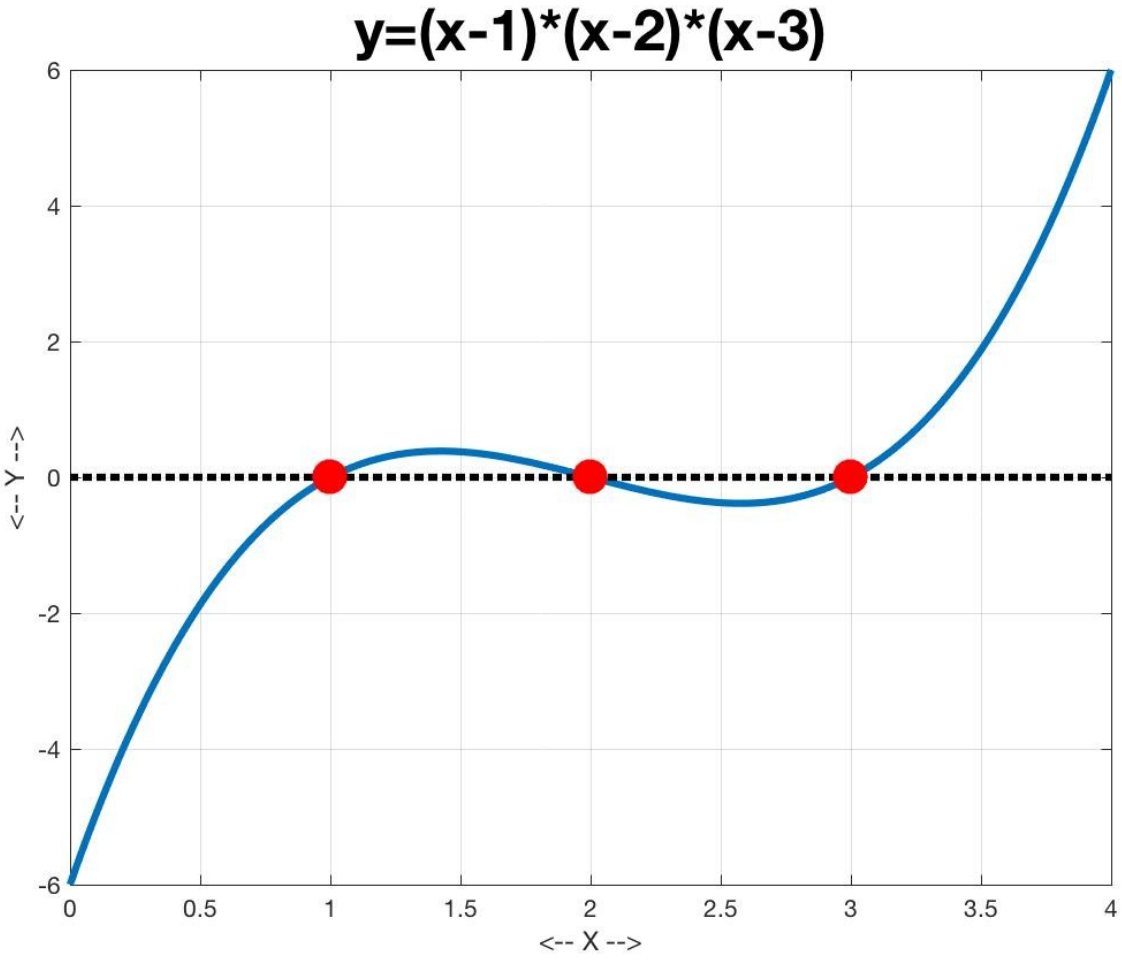
Find all three roots

```
xtol = 0.000001;
a = 0.5;
b = 1.5;
[ x1, a, b ] = bisection3 ( xtol, a, b, @f123 );
fprintf ( 'F(%20.16g) = %g\n', x1, f123 ( x1 ) );

xtol = 0.000001;
a = 1.5;
b = 2.5;
[ x2, a, b ] = bisection3 ( xtol, a, b, @f123 );
fprintf ( 'F(%20.16g) = %g\n', x2, f123 ( x2 ) );

xtol = 0.000001;
a = 2.5;
b = 3.5;
[ x3, a, b ] = bisection3 ( xtol, a, b, @f123 );
fprintf ( 'F(%20.16g) = %g\n', x3, f123 ( x3 ) );
```

We found the roots!



Go Faster with Secant Method

Every step of the bisection method cuts the interval [A,B] in half; we keep doing this until the interval is no bigger than XTOL.

So if we start with B-A=1, and our XTOL is 0.000001, we are guaranteed to need 20 bisection steps, no matter what function we are working with.

This is true even for f(x)=4*x-3!

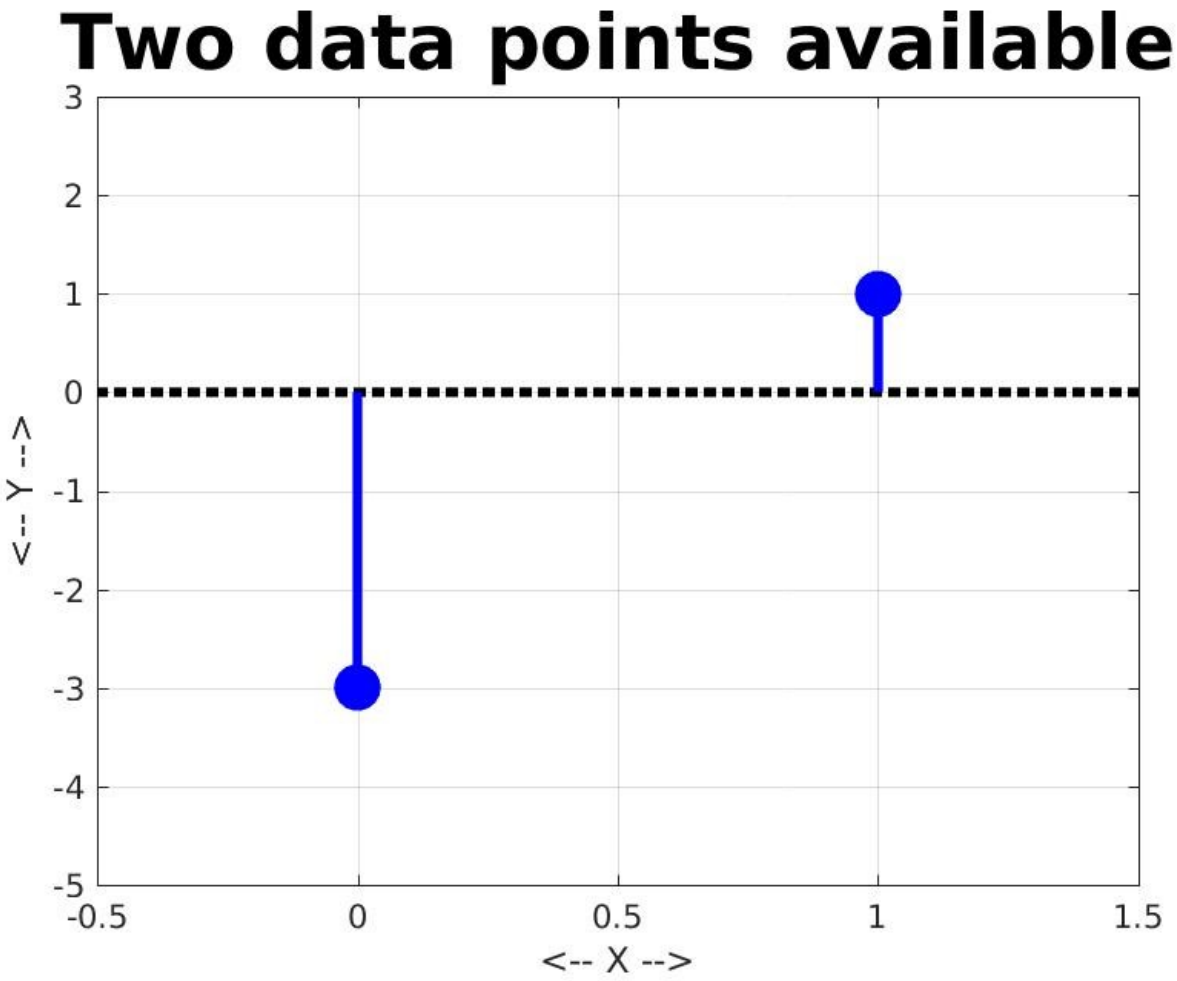
The Idea: Approximate f(x) by a line

The secant method assumes that at points A and B, we have sample values of the function FA=F(A) and FB=F(B).

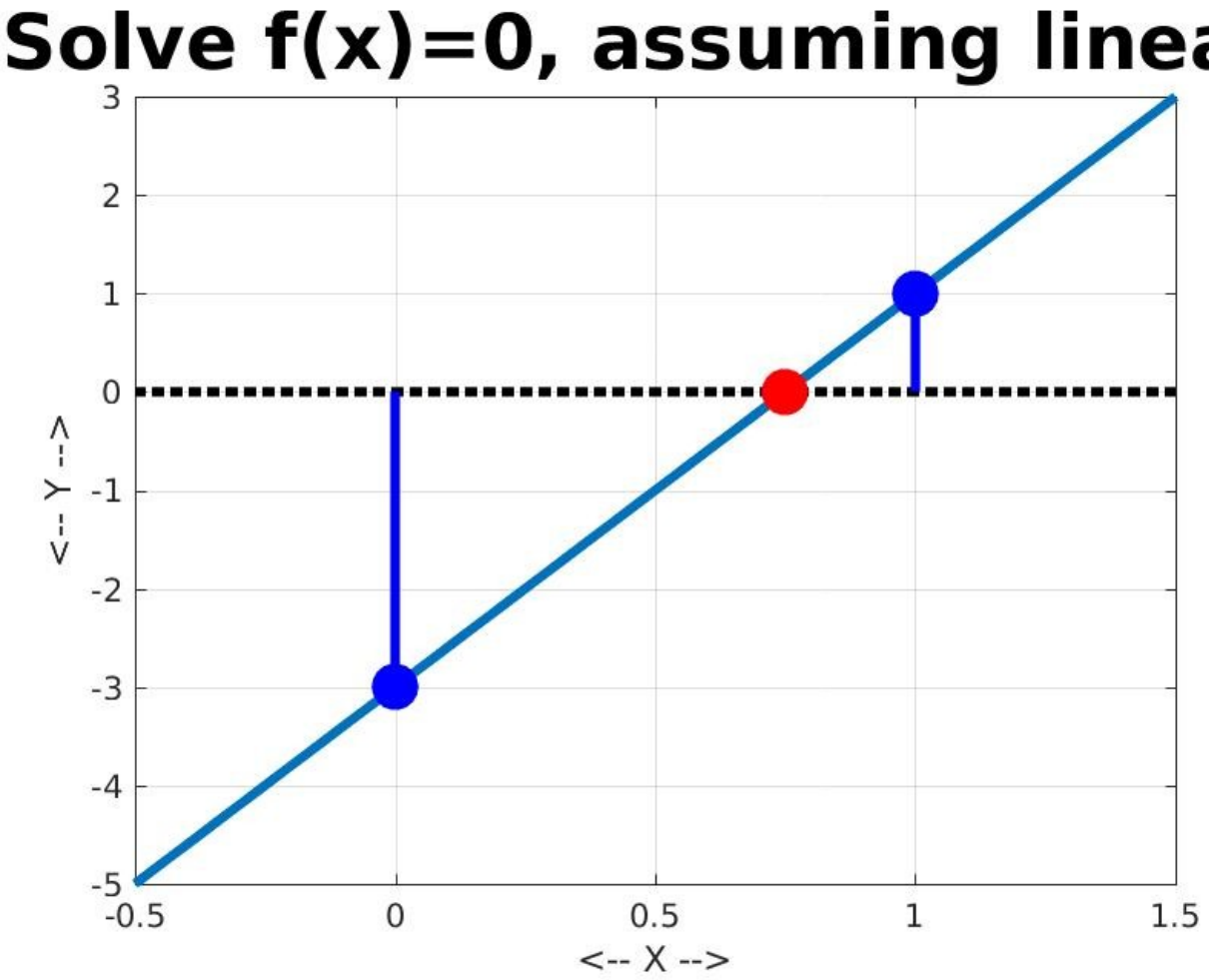
It guesses that nearby, the function is approximately a straight line.

It figures out where the straight line would cross the X axis, and goes there!

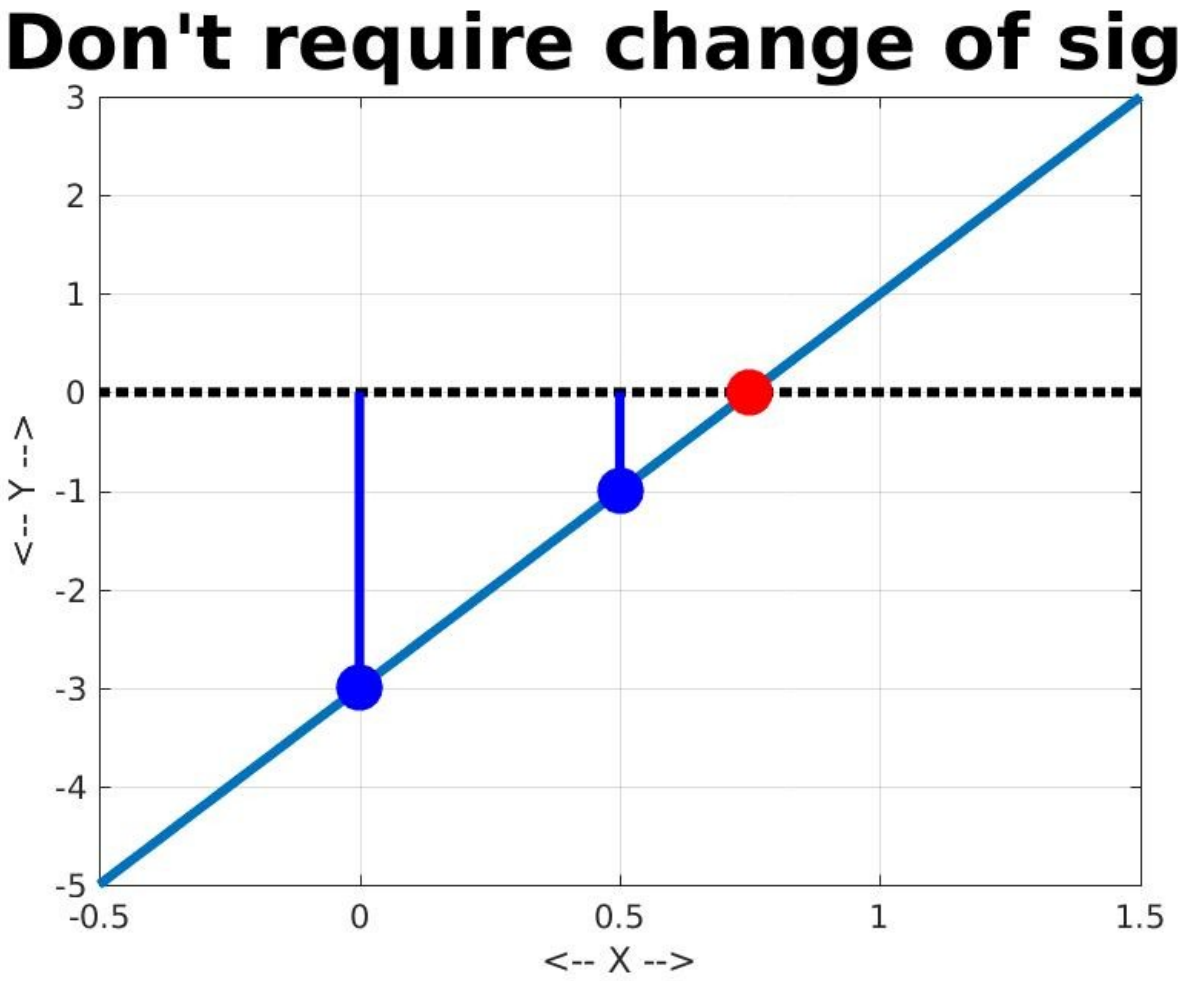
We have two points



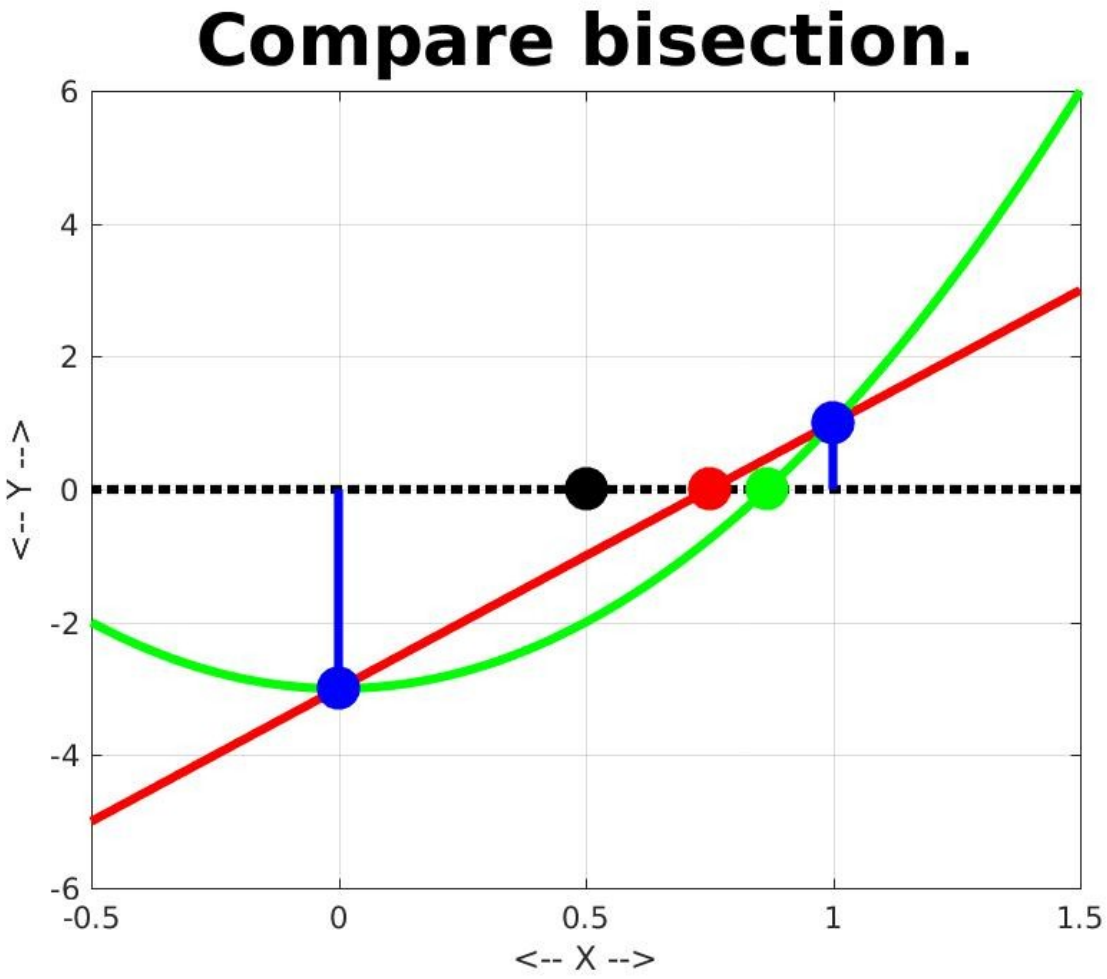
Where would the line cross?



Can try this even without change-of-sign



Can be better than bisection



Computational details

Suppose f(a) = fa, f(b) = fb are the data.

The line through the data has formula:

$$y = f(a) + (f(b) - f(a)) * (x - a) / (b - a).$$

Set y to 0, and solve for x:

$$x = a - f(a) * (b - a) / (f(b) - f(a))$$

The value x is the secant method's estimate for the zero of f(x).

Example

f(x) = 4x^2 – 3

a = 0, f(a) = -3

b = 1, f(b) = 1

Exact zero is at x = 0.866025...

x = a - f(a) * (b – a) / (f(b) - f(a))

= 0 - (-3) * (1 – 0) / (1 - (-3))

= 3/4 = 0.75

Secant error is about 0.116

1 bisection step would have estimated x= 0.5.

Bisection error would be 0.366...

Secant iteration

We can repeat the secant step, just as we do bisection. We started with two points, A, and B, and we computed a new point X.

To repeat the secant step, we can "retire B":

B = A;

A = X;

Use new A and B to take new secant step.

Stopping test

The secant method does not require a change of sign interval, so we can't assume that |B-A| measures how close we are to the correct answer.

Instead, it is common to track |F(X)|, that is, to set a function value tolerance FTOL, and stop the iteration as soon as |F(X)| < FTOL.

Secant method interface

Input:

ftol, a function value tolerance

a, b, the initial points.

secant_max, the maximum number of steps

f, the name of the function file

Output:

x, the estimated zero;

secant_num, the number of steps taken.

secant.m

```
function [ x, secant_num ] = secant ( ftol, a, b, secant_max, f )
  secant_num = 0;
  while ( secant_num < secant_max )
    x = a - f(a) * ( b - a ) / ( f(b) - f(a) );
    if ( abs ( f(x) ) <= ftol )
      return
    end
    b = a;
    a = x;
  end
  return
end
```

Compare Bisection vs Secant

Linear

20 bisections, F(0.749999523162841 8) = -1.90735e-06

1 secant, F(0.75) = 0

Quadratic

20 bisections, F(0.866025447845459) = 3.05264e-07

6 secants, F(0.866025402561513 5) = -8.47267e-09

Cos(x)-x

20 bisections, F(0.739084720611572 3) = 6.90538e-07

4 secants, F(0.739084670239384 2) = 7.74842e-07

The Efficiency Question

If we measure cost in terms of function evaluations, then each step of the secant method must cost at least one function value, since we need to compute F(X).

As I have written secant.m, however, each step uses 4 function evaluations, because of the way I wrote the formula

```
x = a - f(a) * ( b - a ) / ( f(b) - f(a) );
```

For efficiency, I would save function values in variables FA and FB.

secant.m

```
function [ x, secant_num ] = secant ( ftol, a, b, secant_max, f )
  secant_num = 0;
  fa = f(a);  fb = f(b);      <- Initialize FA and FB.
  while ( secant_num < secant_max )
    x = a - fa * ( b - a ) / ( fb - fa );
    fx = f(x);                <- Only ONE call to F inside the loop.
    if ( abs ( fx ) <= ftol )
      return
    end
    b = a;  fb = fa;          <- Update FA and FB.
    a = x;  fa = fx;
  end
  return
end
```

Failure #1

If we happen to pick A and B so that F(A) = F(B), we have a numerical catastrophe!

```
x = a - f(a) * ( b - a ) / ( f(b) - f(a) );
```

Even if f(b) and f(a) are not equal, but very close, a huge stepsize may result.

Example: f(x)=x^2, a=-1, b=+1.

These data points make f(x) look like a flat line, when in fact it's a parabola.

Failure Demonstration #1

```
ftol = 0.00001;
a = -1.0;
b =  0.9999;
secant_max = 1;   <- Let's just see first step!
[x,secant_num] = secant(ftol,a,b,secant_max, @xsquared)
```

.

Failure #2:

If we don't ask for a change of sign interval, then the secant method can chase a zero that isn't there.

Example:

f(x) = e^x, a = 0, b = 1

This function has no zero. However, the secant method will move to the left, taking larger and larger steps, because its straight line model crosses the x axis.

Failure Demonstration #2

```
ftol = 0.00001;
a = 0.0;
b =  1;
secant_max = 1;   <- Let's just see first step!
[x,secant_num] = secant(ftol,a,b,secant_max, @exponential)
```

.

Safe & Effective: "False Position"

The secant method can be more effective than the bisection method, because it's better at coming close to the location of the zero...if a straight line is a good model for the shape of the function.

The bisection method is safe because we know a zero lies somewhere in our bisection interval, and we gradually squeeze the interval to a small size.

We can try a combination of the two methods that gets the advantages of both. This method has the somewhat silly name of "false position", and is sometimes known by its Latin name of "Regula Falsi".

Method of False Position

1) Begin with a change of sign interval, [A,B].
2) While XTOL < B-A
   2.1) Compute secant approximation to the zero:
   X = A - F(A) * (B – A) / (F(B) - F(A))
   Because F(A) and F(B) have opposite signs,
   A < X < B is **guaranteed**.
   2.2) if |F(X)| <= FTOL, you can stop early.
   2.3) depending on sign of F(X), X replaces A or B.
3) Because B-A <= XTOL, you can stop now.

We probably need to include a STEP_MAX check too.

false_position.m

```
function [ x, a, b, step_num ] = false_position ( xtol, ftol, a, b, step_max, f )

  (Check for change of sign interval omitted)

  step_num = 0;

  while ( xtol < b - a );

    (check for number of steps omitted)

    x = a - f(a) * ( b - a ) / ( f(b) - f(a) );
    if ( f(x) < ftol )
      return
    end

    if ( sign ( f(x) ) == sign ( f(a) ) )
      a = x;
    else
      b = x;
    end

  end

  return
end
```

How MATLAB does it

MATLAB provides a function fzero() which can seek a zero of a function.

The simplest call to fzero looks like this:

```
x = fzero ( @fun, x0 );
```

where "fun" is the name of the file defining the function, and x0 is a single starting value for the iteration.

Example: f(x)=cos(x)-x

```
>> format long
>> x = fzero ( @cosxx, 1.0 )
x =

   0.739085133215161

>> cosxx(x)

ans =

    0
```

Specifying an interval

The input argument x0 can actually be a list of two values.

If length(x0)==2, the two values must represent a change-of-sign interval.

MATLAB will search for a zero within this interval.

Example: f123=(x-1)*(x-2)*(x-3)

```
x = fzero ( @f123, [ 0.5, 1.5 ] );
x = 1
f123(x) = 0

x = fzero ( @f123, [ 1.5, 2.5 ] );
x = 2
f123(x) = 0

x = fzero ( @f123, [ 2.5, 3.5 ] );
x = 3
f123(x) = 0
```

Must be a Change-of-Sign Interval

If the x0 input to fzero is a pair of values, they must represent a change-of-sign interval:

```
>> x = fzero ( @f123, [0.0,0.5] )
Error using fzero (line 290)
The function values at the interval endpoints must differ in sign.
f123(0.0) = -6
f123(0.5) = -1.875
```

What if there is no zero?

```
x = fzero ( @exponential, 0.0 );
x = -9.26819000236831 9e+02

>> exponential(x)

ans = 0
```

FZERO is "fooled" because:

a) we didn't give a change of sign interval;

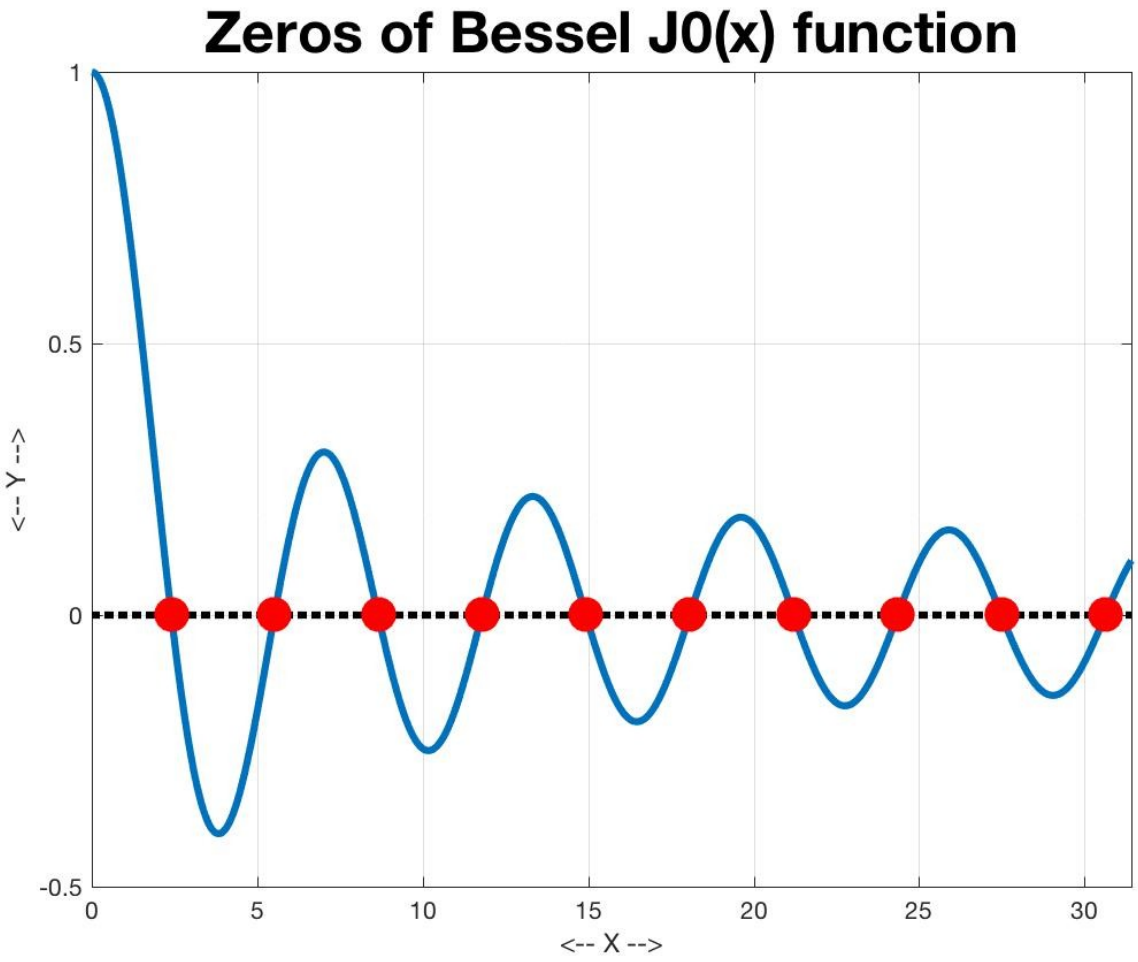
b) e^(-926) is really really small.

c) FZERO includes a function value tolerance.

```
J0 = @(x) besselj ( 0, x );      <- a "one line" function;
for n = 0 : 10
  z(n) = fzero ( J0, [ (n-1)*pi, n*pi] );
end
x = linspace ( 0.0, 10.0*pi, 501 );
y = J0(x);
plot ( x, y, '-', ...
       x, 0*y, 'k:', ...
       z, zeros ( 1, 10 ), 'o', ...
       'LineWidth', 3, 'Markersize', 50 );
axis ( [ 0.0, 10.0 * pi, -0.5, 1.0 ] );
```

A Peculiar Exampl

The plot



"Anonymous Functions"

As a convenience, MATLAB has a way to write a one-line version of a function.
Instead of

```
function value = J0 ( x )
  value = besselj ( 0, x );
  return
end
```

you can write

```
J0 = @(x) besselj ( 0, x );
```

The form is:

```
function_name @ ( input  ) formula involving input;
```

Then you can use J0 and the expression J0(X) as though you had written the full function file.

Homework #7: Due October 27

hw041: use the bisection method to find a zero of the function f(x)=x^3-2x-5.

hw042: Use bisection to find three zeros of a cubic function.

hw043: Use bisection to solve for the value of the golden ratio, phi.

(And homework #6 is due October 20)