Intro Math Problem Solving
October 17

A Mountain Climbing Question

The Zero Finding Problem

The Bisection Algorithm

A Bisection Script

Using a Function Name as Input

Three Tries at a Bisection Function

Solve Some Test Cases

Reference

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.

A Mountain Climbing Question

Same spot, same time?

Very early Monday morning, a team began climbing a mountain, and reached the top sometime before midnight.

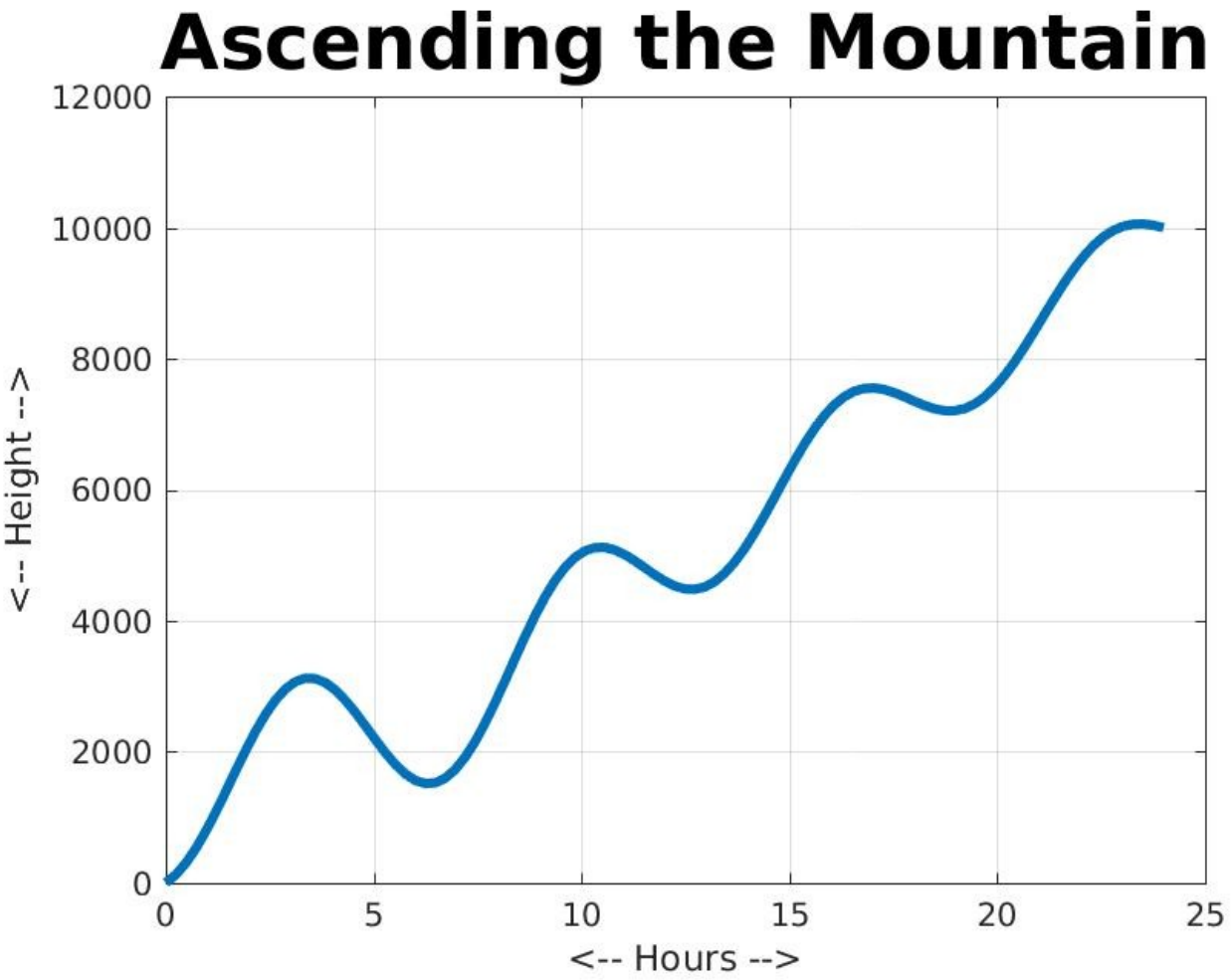
Very early Tuesday morning, they started back down, reaching the bottom before midnight.

At one point going down, a climber said, "We were exactly at this spot, at exactly this time, yesterday while going up."

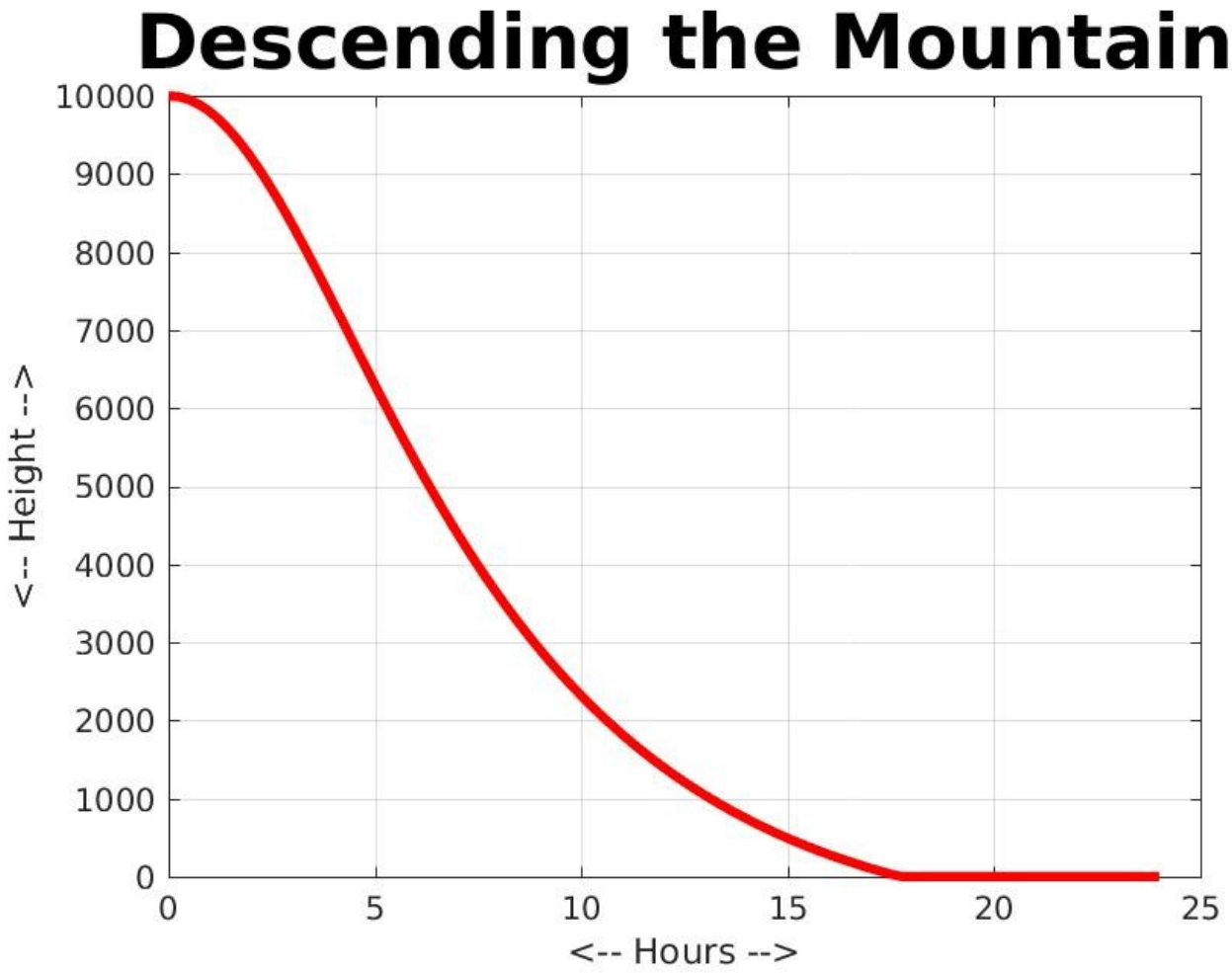
Another climber said, "I think that's so unlikely."

Another climber said, "That must always happen!"

Going up


Ascending the Mountain
12000
10000
8000
6000
4000
2000
0
<-- Height -->
0
5
10
15
20
25
<-- Hours -->

Going Down


Descending the Mountain
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
<-- Height -->
0
5
10
15
20
25
<-- Hours -->

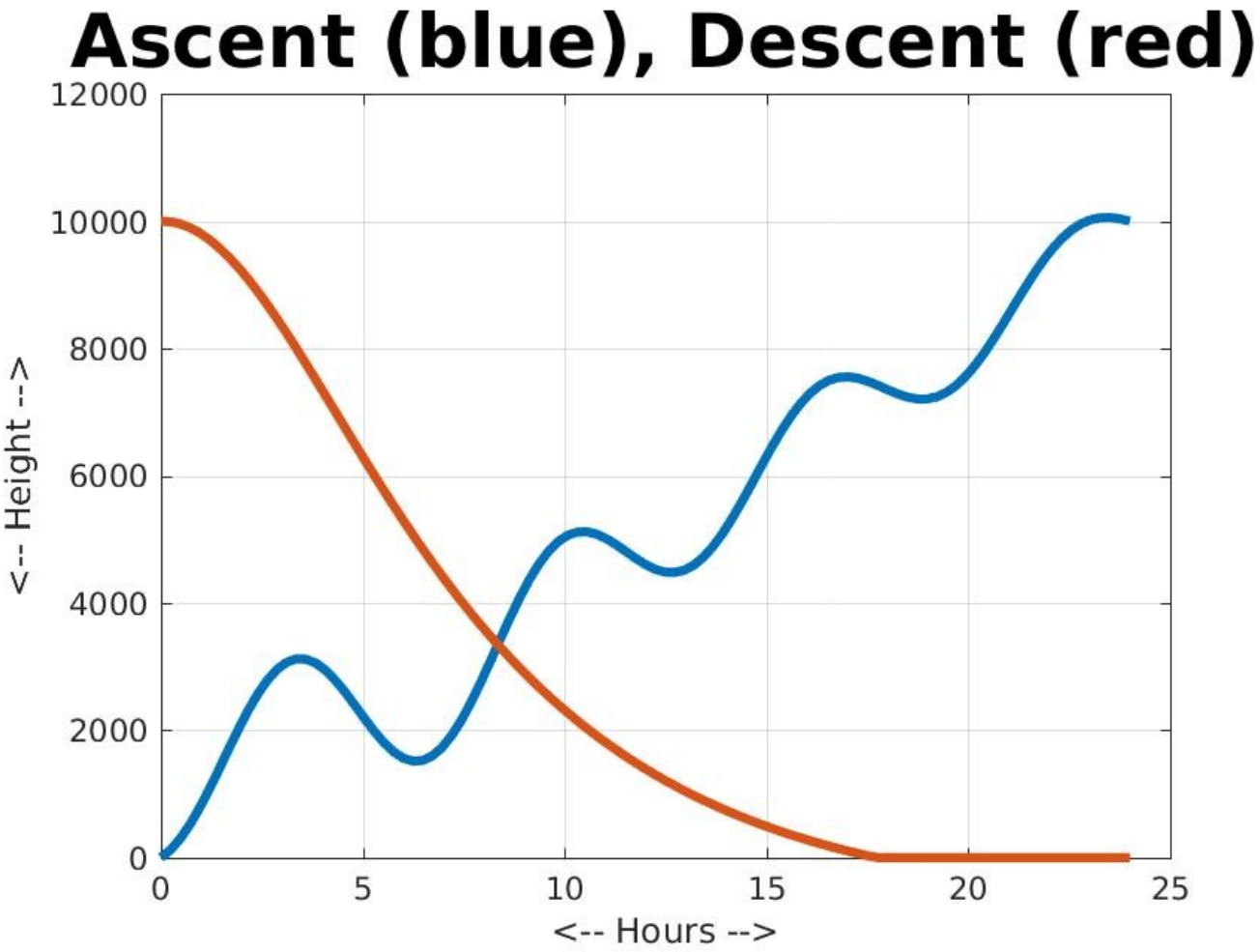
It must always happen!

There must be at least one spot that both the ascending and descending parties will reach at the very same time.

Suppose that, on Wednesday, we put one team at the top and one at the bottom, and then one team descend and one team ascend, exactly following the schedules of the Monday and Tuesday climbers.

At some point and time, the two teams must cross paths.

X Marks the Spot



Math: There is an answer.
Computing: What is it?

So now we know that there is a time of day so that the Monday ascenders and the Tuesday descenders were at the same place. In mathematics, this is called an "existence" proof...there is a solution...although we haven't thought about how to compute it.

If we can characterize our problem more clearly, and find some similar examples, then perhaps we can come up with a computational method to actually find (approximate) solutions.

In the mountain case, "*to within one minute, when did the ascenders and descenders reach the same point?*"

Problem: Find h that makes equation true

A mathematician will look at the mountain climbing problem and think:

ascent(h) is a function that tells me my height at any time h during the ascent;

descent(h) does the same for descent.

I am looking for a value h so that

ascent(h) = descent(h)

that is, "same time(h), same place".

Related Problems

What does each of three people get when sharing 17 pounds of sugar?
"find x so that 3*x = 17"

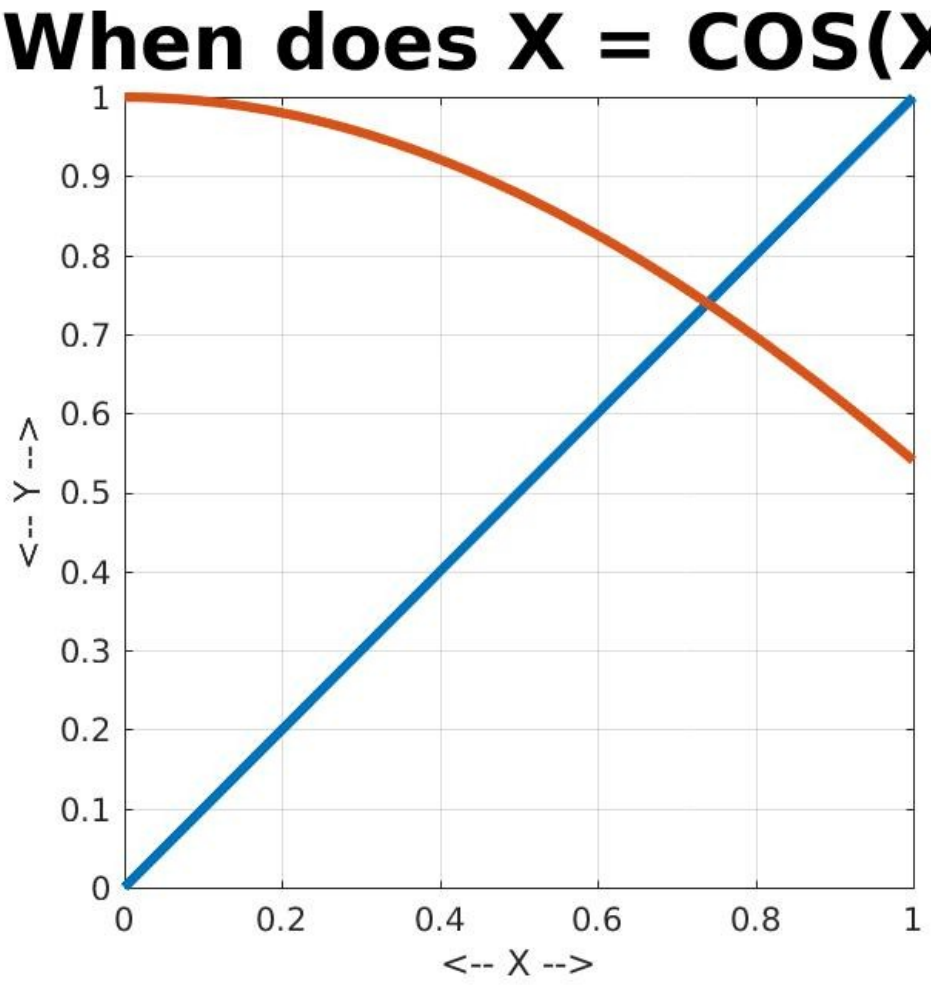
What is sqrt(2017)?
"find x so that x^2 = 2017"
We want a **number**, not the expression sqrt(2017)!

When will my child weight 1000 pounds?
"find n so that "theron(n) = 1000".

After how many seconds t will a falling penny reach the center of the earth?
"find t so that 2080-16t^2=-20900000".

When does the cosine curve cross the line y=x?
"find A so that cos(A) = A".

Let's Look at the X=COS(X) Question



Try #1 to solve X = COS(X)

Make a plot showing both curves:

y = x

y = cos(x)

Zoom in repeatedly and focus on the crossing point.

Disadvantages:

* requires estimating numbers visually;
* not automatic; user must do it;
* plot is made up of straight line segments so results are of limited accuracy.

Try #2 to solve X =COS(X)

Just "turn the dial", trying different values of X.

X	COS(X)	
1	0.5403	(try smaller X)
0.9	0.6216	(getting closer)
0.75	0.7317	(got first digit!)
0.74	0.7385	
0.73	0.7452	(now COS(X) is bigger!)
0.735	0.7418	
0.737	0.7405	
0.738	0.7398	(got two digits to match!)
…	…	

Disadvantages?

- Again, not automatic.
- Not really clear how to pick the next trial point.

Try #3 to solve X=COS(X)

OK, probably can't get exact value. But let's pick a bunch of X values in [0,1], and measure the magnitude of the error, |x-cos(x)|, and find the value of X which minimizes this error.

```
x = linspace ( 0.0, 1.0, 101 );
e = abs ( x – cos(x) );
[min_value, min_index ] = min ( e );
x_min = x(e_index)
cos(x_min)
```

x_min = 0.7400
cos(x_min) = 0.7385
and we know best value is probably between 0.73 and 0.75!

Repeat with `x = linspace ( 0.73, 0.75, 101 );`

x_min = 0.7390
cos(x_min) = 0.7391 ← Already three digits right!
and we know best value is probably between 0.7388 and 0.7392.

Advantages: Semiautomatic.
Disadvantages: we compute a LOT of function values.
Could be fooled by a "near" zero, such as y=(x+0.0000001)^2

Time to Lay Out a Plan

All our problems can be thought of as problem A:

A) find a value x so that f(x)=0.
(When does polynomial f(x)=x^3-2*x-5=0?)

B) "When does g(x) = h(x)?"
(Mountain climbing example)
Define f(x) = g(x) – h(x), solve f(x)=0.

C) "When does g(x) = value?"
(Theron baby-weight example)
Define f(x) = g(x) – value, solve f(x)=0.

Problem A is called "the zero-finding problem". But now we know all these problems can be called "the zero finding problem" if we think about them that way.

Computer Arithmetic

Mathematically, seek x so that $f(x) = 0$.

In computer arithmetic, the exact solution x might not exist. And even if x exists (is a number representable on the computer), $f(x)$ might not come out to 0, because of rounding errors.

So instead of $f(x)=0$, we might seek x so

$|f(x)| < TOL$

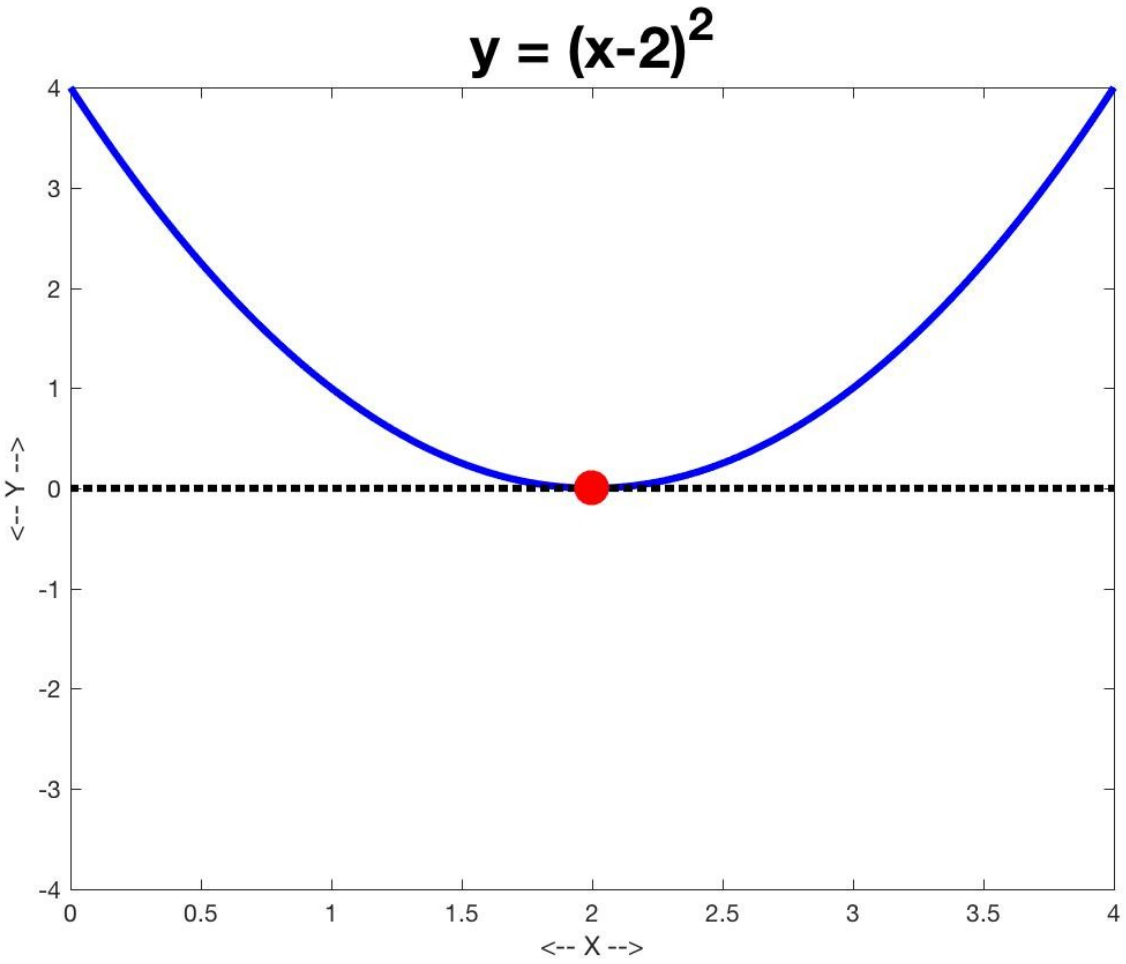
Change of Sign

Seeking |f(x)|<TOL can be deceptive. For example, e^x is NEVER zero, but the exponential function gets very small for strongly negative x.

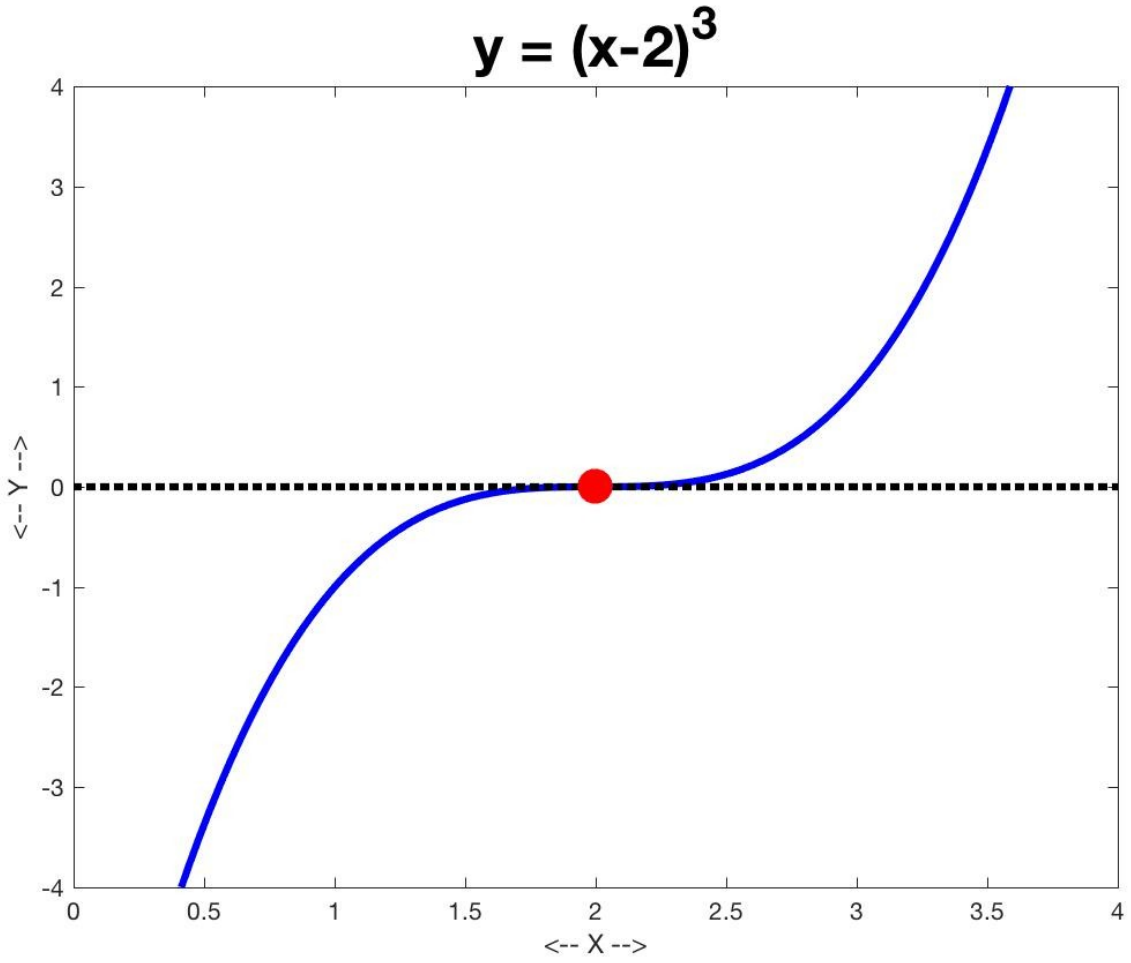
A better test begins by stating we can only be confident about finding a zero at x if the function actually changes sign there, that it goes from positive to negative (or the other way around).

This means we are not going to bother trying to catch zeros where the function just "grazes" the x axis, such as f(x) = (x-2)^2.

This function only "grazes" the x axis



This function crosses the x axis



Continuity Guarantees Solution

Mathematics has an intermediate value theorem:

If f(x) is a continuous function on the interval [a,b], then for every value **v** between f(a) and f(b), there is an x between a and b so that f(x)=v.

In particular, if f(a) is negative and f(b) is positive, then somewhere between a and b, there must be an x so that f(x)=0!

Change of Sign Interval

If a function actually crosses the x axis, then we know there is a solution f(x)=0, but also, we know that nearby the function is positive on one side and negative on the other.

Over a small interval [A,B], the function changes sign, and a solution to f(x)=0 is somewhere inside.

Even if we can't find x exactly, we can decrease the size of [A,B] so that we fence in the value of x to very good approximation.

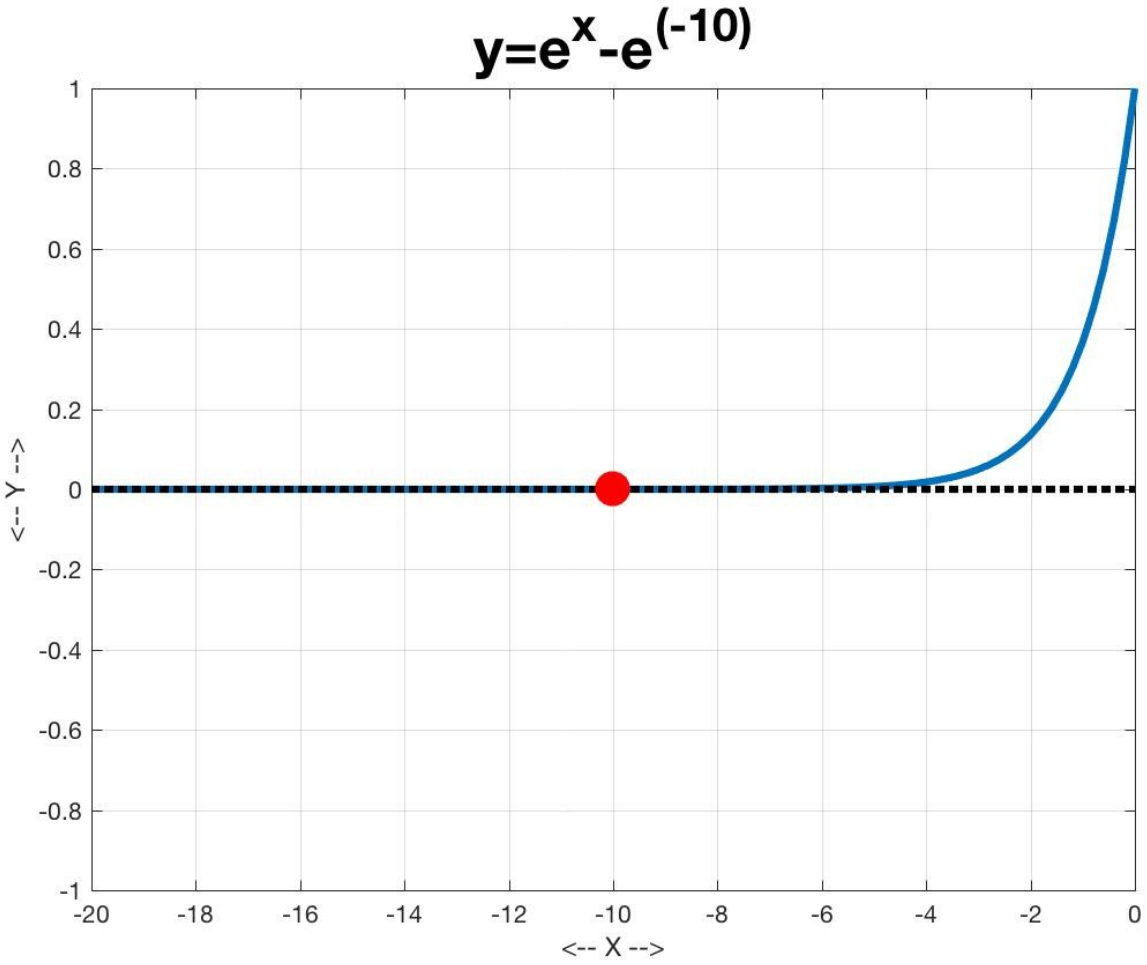
Two Crazy Extremes

We will hope to find small intervals [A,B] containing a solution, and we will hope to find approximate zeros x so that |f(x)| is very small.

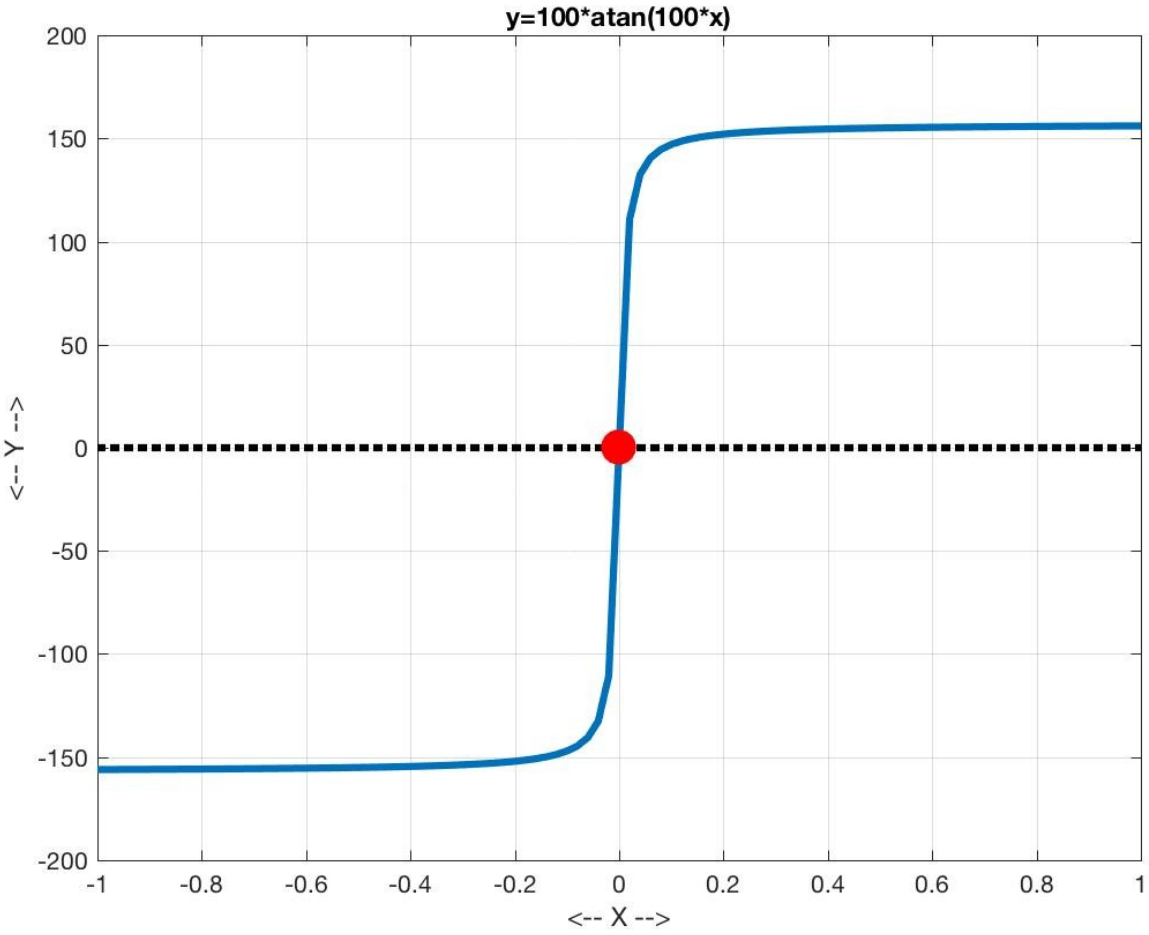
One bad case occurs if the function f(x) is very small over a very wide range, such as e^x for negative x.

Another bad case occurs if the function has very large magnitudes near the zero.

|f(x)| < TOL interval very wide



|f(x)| < TOL interval very narrow



A Computable Answer

The mathematical problem seeks a number x so that f(x)=0 exactly.

The computational problem starts with a change of sign interval [A,B], and seeks to reduce the width of [A,B] to a small value, and to produce an estimated zero x so that |f(x)| is small.

Computationally, we may not be able to do both.

F(X) = COS(X) - X with Change of Sign

X	F(X)	Positive F(X)	Negative F(X)	Width
0	1	F(0) = 1		?
1	-0.4597		F(1.0) = -0.4597	1.0
0.9	-0.2784		F(0.9) = -0.2784	0.9
0.75	-0.0183		F(0.75) = -0.0183	0.75
0.73	0.0152	F(0.73) = 0.152		0.02
0.735	0.0068	F(0.735) = 0.0068		0.015
0.737	0.0035	F(0.737) = 0.0035		0.013
0.738	0.0016	F(0.738) = 0.0016		0.012
0.74	-0.0015		F(0.74) = -0.0015	0.002
0.739	0.0001	F(0.739) = 0.0001		0.001

Change of Sign Information

We still haven't found an exact solution for f(x)=cos(x)-x=0, but as soon as we found a change-of-sign interval, we knew that a solution must be between x=0 and x=1.

By updating our change of sign interval, we now know that a solution must be between x=0.739 and x=0.740.

Also, we can guess (this is only a guess) that, for all the x's in this interval, the function values are

$-0.0015 \leq f(x) \leq 0.0001$

However, this method is still not automatic, because I just picked the next x to investigate by guessing.

A Zero-FindingAlgorithm

To find a zero of a function f(x) within an interval tolerance XTOL or function tolerance FTOL:

Find a change of sign interval [A,B];

As long as XTOL < |B-A|

Choose a point C inside [A,B].

Replace A or B by C (using sign of F(C))

Interval small enough, X = (A+B)/2, stop!

Bisection Method

The simplest way to "choose a point C in [A,B]" is to choose the midpoint:

$C = (A+B)/2$.

C splits the interval into two halves. If the sign of F(C) matches the sign of F(A), we keep the interval [C,B]; otherwise, [A,C].

Thus each step cuts our interval size in half, and we automatically zoom in towards our answer.

Checking the sign

We need to check whether the sign of F(C) matches that of F(A) or F(B).

One way:

```
if ( ( f(c) < 0.0 && f(a) < 0.0 ) ||
     ( f(c) > 0.0 && f(a) > 0.0 ) )
  a = c;
else
  b = c;
end
```

Better:

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

The MATLAB sign() function

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x > 0.0 \\ 0 & \text{if } x == 0.0 \\ -1 & \text{if } x < 0.0 \end{cases}$$

sign(7) = 1;

sign(-3) = -1;

sign(0) = 0;

sign(0.0000014) = 1;

Sketch of Bisection Code

```
while ( xtol < b - a )   % assuming A < B!

  c = ( b + a ) / 2;

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

end

x = ( b + a ) / 2;
```

bisection1.m (more details)

```
xtol = input ( 'Enter tolerance for interval size: ' );
a = input ( 'Enter left  endpoint of change of sign interval: ' );
b = input ( 'Enter right endpoint of change of sign interval: ' );

if ( sign ( f(a) ) * sign ( f(b) ) ~= -1 )
  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;
fprintf ( 'Estimated zero F(%20.16g) = %g\n', x, f(x) );
```

Sample results for f(x)=cos(x)-x

```
>> bisection1
Tolerance for interval size: 1.0e-05
Left  endpoint of change of sign interval: 0
Right endpoint of change of sign interval: 1

Estimated zero F(  0.7390861511230469) = -1.70358e-06
Interval: [  0.7390823364257812,  0.7390899658203125]
Interval width: 7.6294e-06
.
.
.
.
```

A functional version?

bisection1.m is a script.

* The user enters values interactively.
* Variables in the script could affect the user's code;
* The user function must be named "f".

Could we create a function like this:

x = bisection2 (xtol, ftol, a, b)? ← No!

Wait, we need to specify the function too!:

x = bisection2 (xtol, ftol, a, b, f)? ←Not quite!

This ALMOST works. But because "f" is the name of a function, MATLAB requires us to mark it with a special "AT" sign:

x = bisection2 (xtol, ftol, a, b, @f) ← This will work!

Let's understand how a function can be used as input to another function.

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.

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

By using the "@" sign on your input, you can specify the particular function you have in mind. Meanwhile, the function that does the work can behave as though it's working with a function named "f", or whatever temporary name is used for the input.

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

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.

bisection2.m

```
function [ x, a, b ] = bisection2 ( xtol, a, b, f )

  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
```

How to use bisection2.m

```
xtol = 1.0e-5;
a = 0.0;
b = 1.0;

[ x, a, b ] = bisection2 ( xtol, a, b, @cosxx );

fprintf ( 'Estimated zero F(%20.16g) = %g\n', x, cosxx(x) );
fprintf ( 'Interval [%20.16g,%20.16g]\n', a, b );
fprintf ( 'Interval width = %g\n', b - a );
```

```
Estimated zero F(  0.7390861511230469)=-1.70358e-06
Interval [  0.7390823364257812,  0.7390899658203125]
Interval width = 7.6294e-06
```

Try f(x) = polynomial

Let's try bisection on a new function:

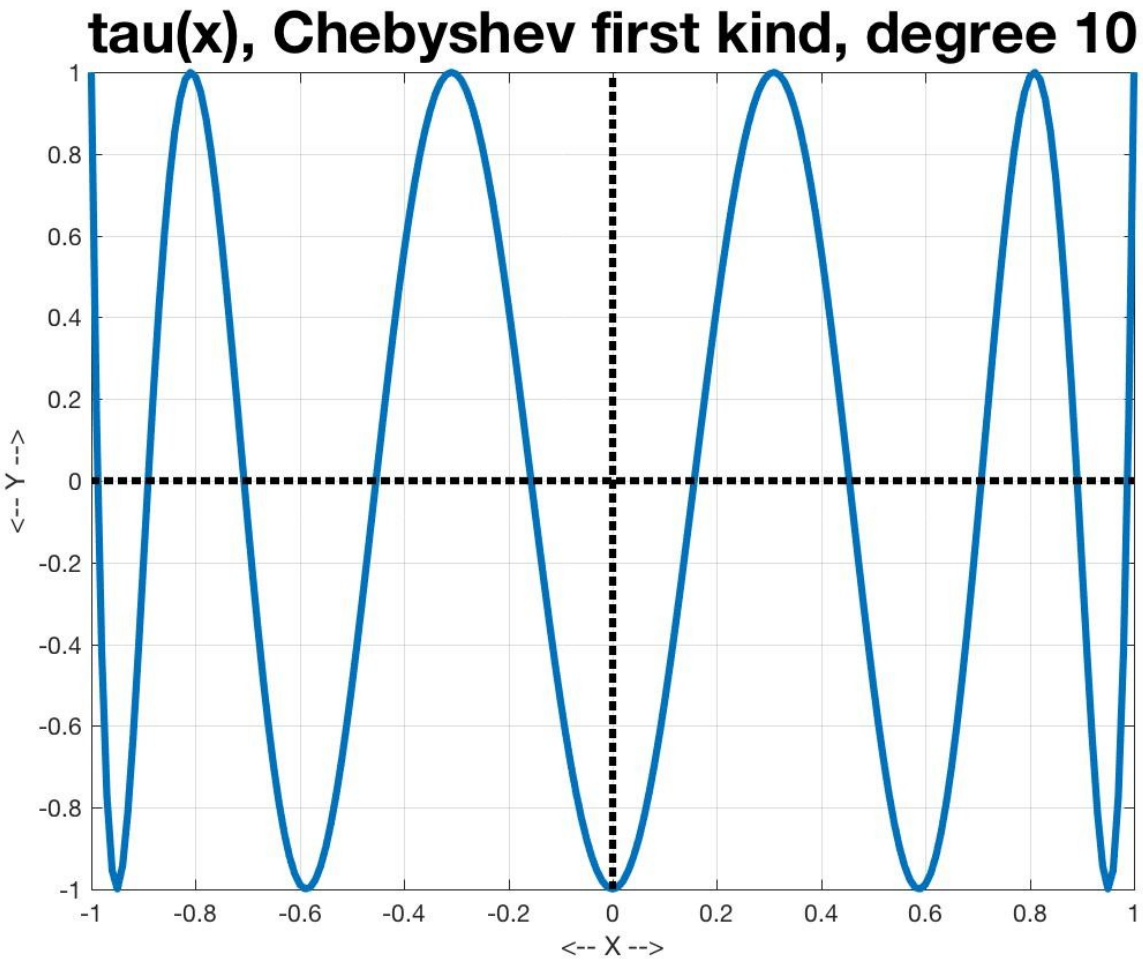
tau(x) = 512 x^10 – 1280 x^8 + 1120 x^6 – 400 x^4 + 50 x^2 – 1

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 )

  step_max = 52;
   step_num = 0;

  while ( xtol < b - a )
    c = ( a + b ) / 2.0;

    step_num = step_num + 1;
    if ( step_max < step_num )
       fprintf ( 'Maximum number of steps exceeded!\n' );
       fprintf ( 'Bisection terminated without satisfying tolerance.\n' );
       x = ( a + b ) / 2.0;
       return
    end

    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.

Some familiar tests

Solve the mountain climbing problem, given the functions ascent.m and descent.m

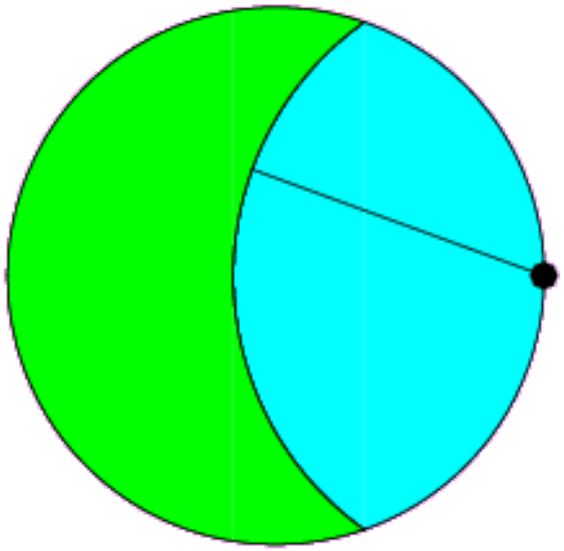
Solve x = sqrt(2017).

At what age n does theron(n)=1000?

At what time t does y(t) = 2080-16t^2 = -20,900,000?

A complicated problem

In a circular field of radius 10 meters, a goat is to be tethered by a rope tied to the fence. How long should the rope be so that the goat can graze on exactly half of the area of the field?



The formula

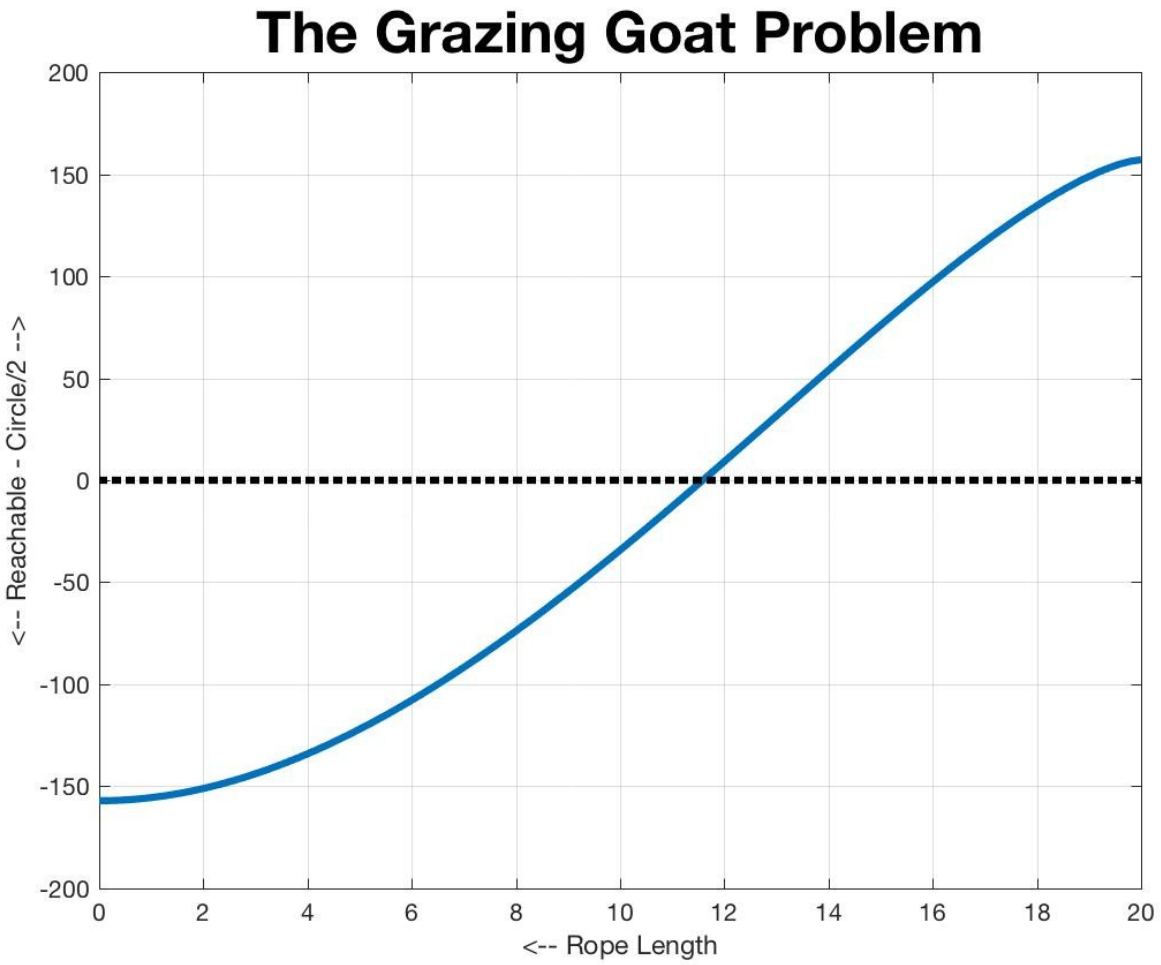
f(x) = Area of circle reachable - 1/2 area of circle

We have to use a formula for the area of the intersection of two circles to get:

f(x) = - x/2 * sqrt(400-x^2) + (x^2-200) * arccos(x/20) + 100 pi / 2

For x = 0 the function is negative, and for x = 20 the function is positive, so we have a change of sign interval.

Looking for f(x) = 0



More, Better Zero Finders?

Recall that our zero finder algorithm outline had the statement "pick a value within the interval [A,B]".

We got the bisection method by making the choice "pick the midpoint", but other choices are possible.

For example, if f(a) is 0.001 and f(b) is -0.16, then it makes sense to guess that the zero is much closer to a than to b.

We will look at this idea next time, and then see how MATLAB has some powerful tools for the zero finding problem.