

1.00 Lecture 19

October 24, 2005

Numerical Methods: Root Finding

Remember Java Data Types

Type	Size (bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2,147,483,648 to 2,147,483,647
long	64	-9,223,372,036,854,775,808L to 9,223,372,036,854,775,807L
float	32	+/- 3.4E+38F (6-7 significant digits)
double	64	+/- 1.8E+308 (15 significant digits)
char	16	ISO Unicode character set
boolean	1	true or false

Remember Numerical Problems

Problem	Integer	Float, double
Zero divide (overflow)	Exception thrown. Program crashes unless caught.	POSITIVE_INFINITY, NEGATIVE_INFINITY
0/0	Exception thrown. Program crashes unless caught	NaN (not a number)
Overflow	No warning. Program gives wrong results.	POSITIVE_INFINITY, NEGATIVE_INFINITY
Underflow	Not possible	No warning, set to 0
Rounding, accumulation errors	Not possible	No warning. Program gives wrong results.

Common, "bad news" cases

Packaging Functions in Objects

- Consider writing method that finds root of function or evaluates a function, e.g.,
 - $f(x) = 0$ on some interval $[a, b]$, or find $f(c)$
- General method that does this should have $f(x)$ as an argument
 - Can't pass functions in Java
 - Wrap the function in an object instead
 - Then pass the object to the root finding or evaluation method as an argument
 - Define an interface that describes the object that will be passed to the numerical method

“Function Passing” Example

```
// Scope is public, so it must be in its own file
public interface MathFunction {
    public double f(double x); }

```

```
// Place this class its own file
// FuncA implements the interface
class funcA implements MathFunction {
    public double f(double x) {
        return x*x - 2;    } }

```

Exercise: Passing Functions

- **Write interface MathFunction2 in its own file**

```
public interface MathFunction2 {
    public double f(double x1, double x2);
}

```

- **In a second file, write a class that implements the interface for the function $5x_1^2 + 2x_2^3$**

```
class FuncB implements MathFunction2 { ... }

```

Exercise (continued)

- Write a method to evaluate functions that takes a `MathFunction2` object and two doubles as arguments
 - `evalFunc()` returns true if $f \geq 0$, false otherwise
 - It's convenient to make `evalFunc()` static
- ```
class Evaluator {
 boolean evalFunc(MathFunction2 func, double
 d1, double d2){...}
```
- Write a `main()`, in `Evaluator` or outside it, that:
    - Invokes `Evaluator`, passing a `FuncB` object to it, and
    - Outputs the value of the function at  $x_1=2$  and  $x_2=-3$

## General Interface for multivariate functions

```
public interface MathFunctionN {
 public double f(double xArray[]);

 //xArray.length provides the number of
 // variables in f(x)

}
```

## **New in Java 5.0 – Variable argument lists**

```
public interface MathFunctionN {
 public double f(double... xval);
}
```

**This allows calls of the form**

```
f(3.0, 2.0, 1.0)
```

**that the compiler converts in to array with as many members as there are parameters**

```
public class TestMathFunctions {
 public static void main(String[] args) {
 MathFuncVarargs f2 = new AltEvaluator2();
 System.out.println(f2.f(3.0,2.0));
 }
}
```

```
class AltEvaluator2 implements MathFuncVarargs {
 public double f(double... x) {
 return 2.0*x[0]*x[0] -4.0*x[1] + 1;
 }
}
```

## Root Finding

- **Two cases:**
  - One dimensional function:  $f(x) = 0$
  - Systems of equations ( $F(X) = 0$ ), where
    - $X$  and  $0$  are vectors and
    - $F$  is an  $N$ -dimensional vector-valued function
- **We address only the 1-D function**
  - In 1-D, it's possible to bracket the root between bounding values
  - In multidimensional case, it's impossible to bound
- **(Almost) all root finding methods are iterative**
  - Start from an initial guess
  - Improve solution until convergence limit satisfied
  - For smooth 1-D functions, convergence assured, but not otherwise

## Root Finding Methods

- **Elementary (pedagogical use only):**
  - Bisection
  - Secant, false position (regula falsi)
- **“Practical” (using the term advisedly):**
  - Brent's algorithm (if derivative unknown)
  - Newton-Raphson (if derivative known)
  - Laguerre's method (polynomials)
  - Newton-Raphson (for  $N$ -dimensional problems)
    - Only if a very good first guess can be supplied
- **See “Numerical Recipes in C” for methods**
  - Library available on MIT server. Can translate to Java
  - The C code in the book is quite (needlessly) obscure

## Preparing for Root Finding

- **Before using root finding methods:**
  - Graph the equation(s): Matlab, etc.
    - Are they continuous, smooth; how differentiable?
  - Use Matlab, etc. to explore solutions
  - Linearize the equations and use matrix methods to get approximate solutions
  - Approximate the equations in other ways and solve analytically
  - Bracket the ranges where roots are expected

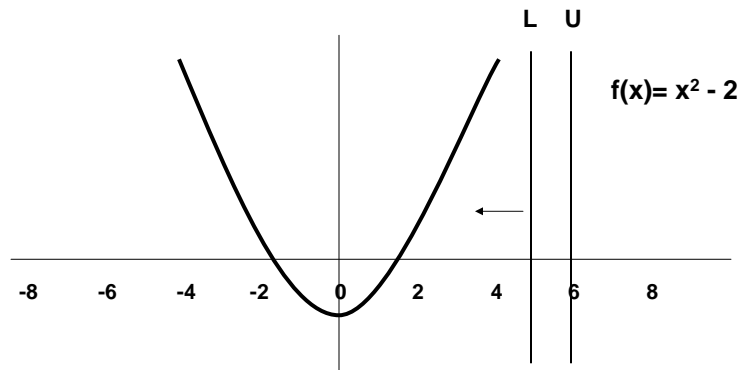
## Some Functions are Evil

- **For fun, look at**
  - Plot it at 3.13, 3.14, 3.15, 3.16;  $f(x)$  is around 30

$$f(x) = 3x^2 - (1/\pi^4)\ln[(\pi - x)^2] + 1$$

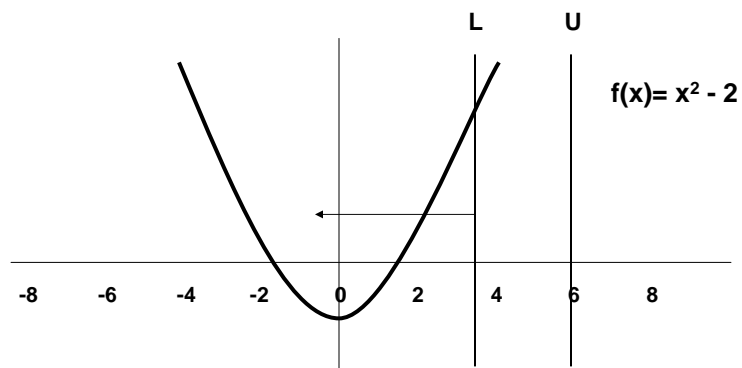
- Well behaved except at  $x = \pi$
- Dips below 0 in interval  $x = \pi \pm 10^{-667}$
- This interval is less than precision of doubles!
  - You'll never find these two roots numerically

# Bracketing



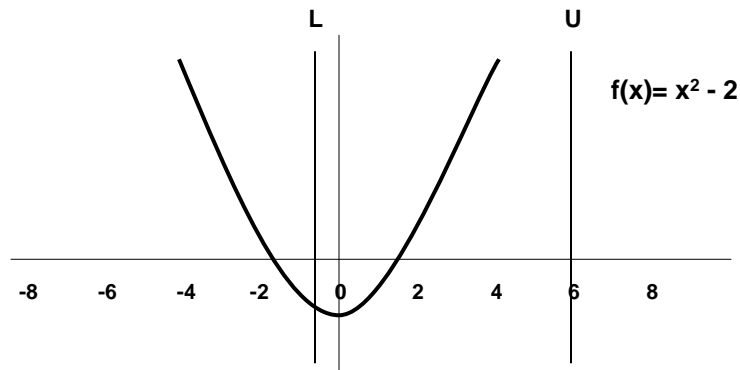
No zero in bracket (though we can't be sure)  
Move in direction of smaller  $f(x)$  value.  
Empirical multiplier of 1.6 to expand bracket size

# Bracketing



Still no zero in bracket (though we can't be sure)  
Move again in direction of smaller  $f(x)$  value.

# Bracketing



Done; found an interval containing a zero

# Bracketing Program

```
public class Bracket {
 public static boolean zbrac(MathFunction func, double[] x){
 // Java version of zbrac() in Numerical Recipes
 if (x[0] == x[1]) {
 System.out.println("Bad initial range in zbrac");
 return false; }
 double f0= func.f(x[0]);
 double f1= func.f(x[1]);
 for (int j= 0; j < NTRY; j++) {
 if (f0*f1 < 0.0)
 return true;
 if (Math.abs(f0) < Math.abs(f1)) {
 x[0] += FACTOR*(x[0]-x[1]);
 f0= func.f(x[0]); }
 else {
 x[1] += FACTOR*(x[1]-x[0]);
 f1= func.f(x[1]); } }
 return false; }
 // No guarantees that this method works!
```

## Bracketing Program

```
// Class Bracket continued
public static double FACTOR= 1.6;
public static int NTRY= 50;

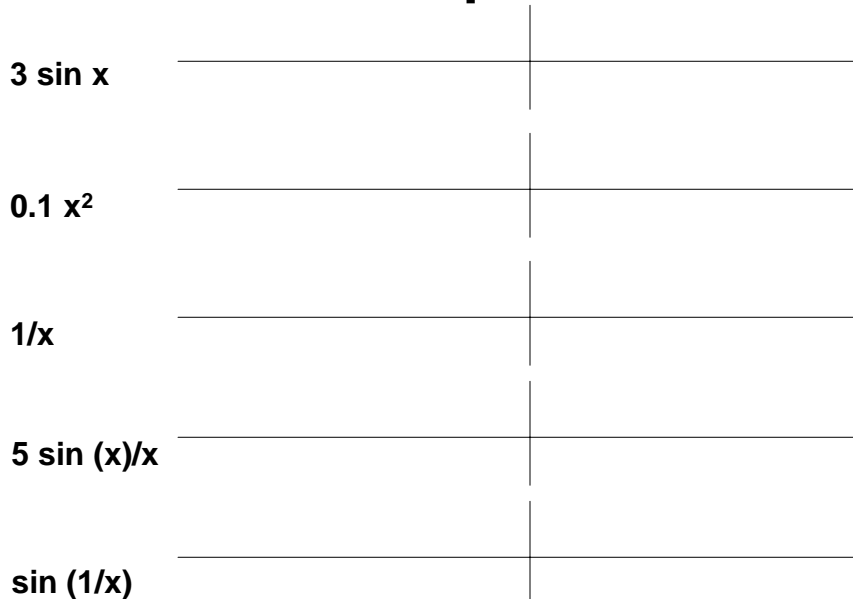
public static void main(String[] args) {
 double[] bound= {5.0, 6.0}; // Initial bracket guess
 boolean interval Found= zbrac(new FuncA(), bound);
 System.out.println("Bracket found? " + interval Found);
 if (interval Found)
 System.out.println("L: "+bound[0]+" U: "+bound[1]);
 System.exit(0);
}
}

// Numerical Recipes has 2nd bracketing program which
// searches subintervals in bracket and records those w/zeros
```

## Exercise: Brackets

- Find intervals where the following functions have zeros or singularities:
  - $3 \sin(x)$
  - $0.1x^2$
  - $1/x$
  - $5 \sin(x) / x$
  - $\sin(1/x)$
- Sketch these roughly
- We'll explore these 5 functions with different root finding methods shortly

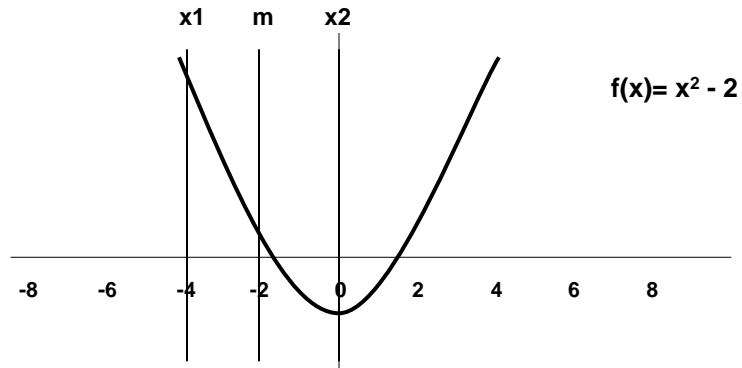
## Exercise: Graph Functions



## Bisection

- **Bisection**
  - Interval given to method must be known to contain at least one root
  - Given that, bisection “always” succeeds
    - If interval contains 2 or more roots, bisection finds one of them
    - If interval contains no roots but straddles singularity, bisection finds the singularity
  - Robust, but converges slowly
  - Tolerance should be near machine precision for double (about  $10^{-15}$ )
    - When root is near 0, this is feasible
    - When root is near, say,  $10^{10}$ , this is difficult
  - Numerical Recipes, p.354 gives a usable method
    - Checks that a root exists in bracket defined by arguments
    - Checks if  $f(\text{midpoint}) == 0.0$
    - Has limit on number of iterations, etc.

# Bisection

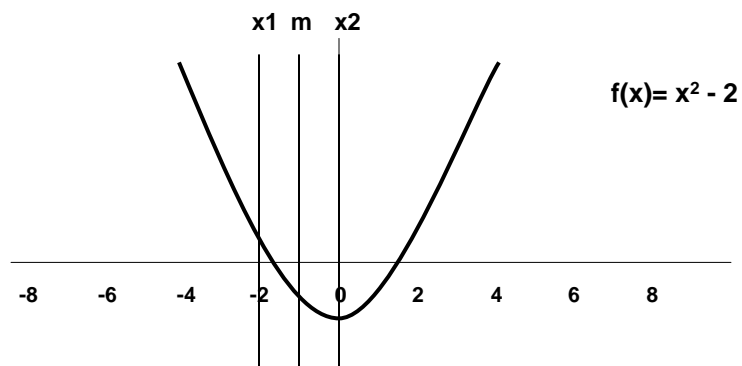


$f(x_1) \cdot f(m) > 0$ , so no root in  $[x_1, m]$

$f(m) \cdot f(x_2) < 0$ , so root in  $[m, x_2]$ . Set  $x_1 = m$

Assume/analyze only a single root in the interval (e.g.,  $[-4.0, 0.0]$ )

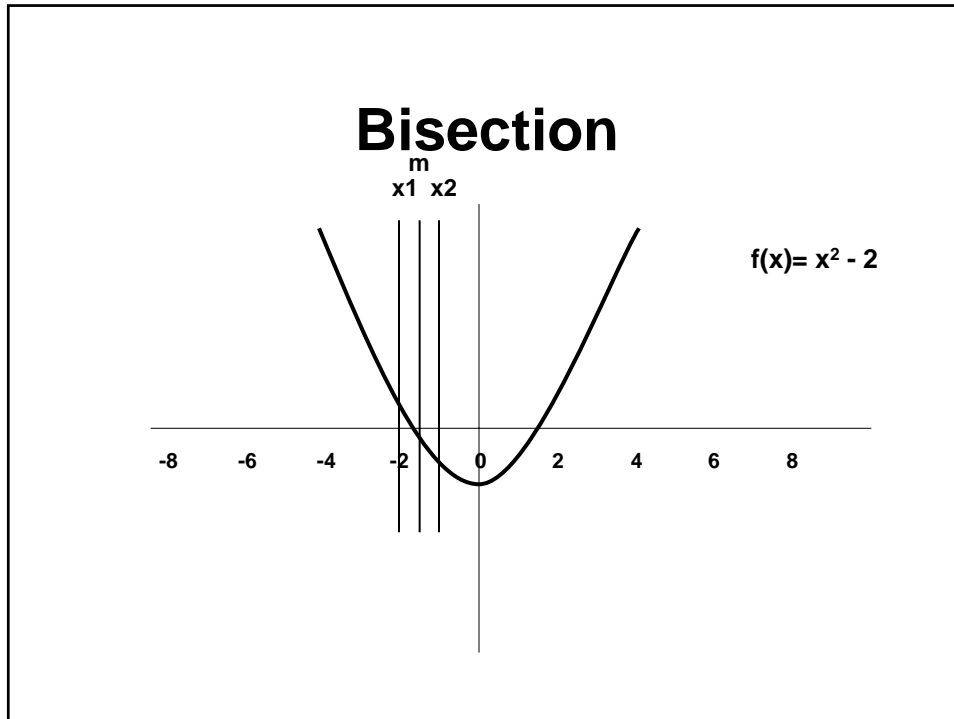
# Bisection



$f(m) \cdot f(x_2) > 0$ , so no root in  $[m, x_2]$

$f(x_1) \cdot f(m) < 0$ , so root in  $[x_1, m]$ . Set  $x_2 = m$

Continue until  $(x_2 - x_1)$  is small enough



## Bisection- Simple Version

```

class FuncA implements MathFunction {
 public double f(double x) {
 return x*x - 2; } }

public class RootFinder1 {
 public static double bisect(MathFunction func, double x1,
 double x2, double epsilon) {

 double m;
 for (m= (x1+x2)/2.0; Math.abs(x1-x2)> epsilon;
 m= (x1+x2)/2.0)
 if (func.f(x1)*func.f(m) <= 0.0)
 x2= m; // Use left subinterval
 else
 x1= m; // Use right subinterval
 return m; }
 }

```

## Bisection-Simple, p.2

```
// main() method in class RootFinder1
public static void main(String[] args) {
 double root= RootFinder1.bisect(
 new FuncA(), -8.0, 8.0, 0.0001);
 System.out.println("Root: " + root);
 System.exit(0); } }
```

## Bisection- NumRec Version

```
public class RootFinder {
public static final int JMAX= 40; // Max no of bisections
public static final double ERR_VAL= -10E10;

public static double rtbis(MathFunction func, double x1,
 double x2, double xacc) {

 double dx, xmid, rtb;
 double f= func.f(x1);
 double fmid= func.f(x2);
 if (f*fmid >= 0.0) {
 System.out.println("Root must be bracketed");
 return ERR_VAL; }
 if (f < 0.0) { // Orient search so f>0 lies at x+dx
 dx= x2 - x1;
 rtb= x1; }
 else {
 dx= x1 - x2;
 rtb= x2; }
 // All this is 'preprocessing'; loop on next page
```

## Bisection- NumRec Version, p.2

```
for (int j=0; j < JMAX; j++) {
 dx *= 0.5; // Cut interval in half
 xmid= rtb + dx; // Find new x
 fmid= func.f(xmid);
 if (fmid <= 0.0) // If f still < 0, move
 rtb= xmid; // left boundary to mid
 if (Math.abs(dx) < xacc || fmid == 0.0)
 return rtb;
}
System.out.println("Too many bisections");
return ERR_VAL;
}
// Invoke with same main() but use proper bracket

// This is noticeably faster than the simple version,
// requiring fewer function evaluations.
// It's also more robust, checking brackets, limiting
// iterations, and using a better termination criterion.
// Error handling should use exceptions, covered later!
```

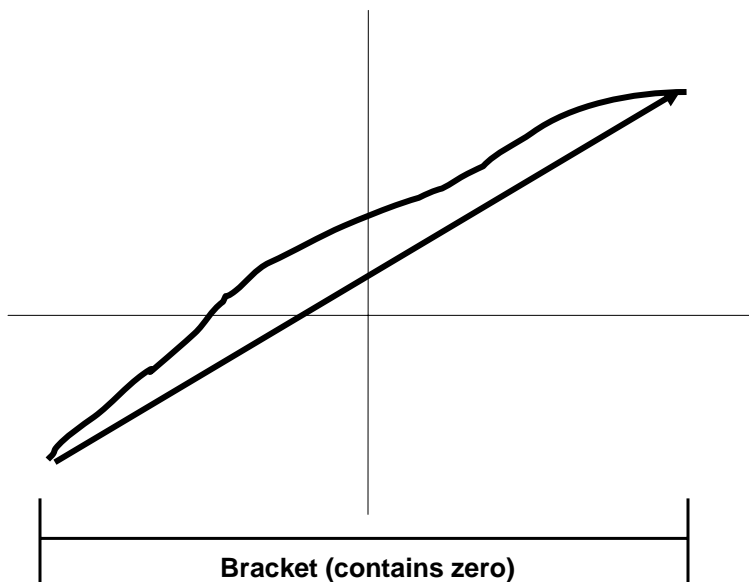
## Exercise: Bisection

- Download Roots.java from Web site
- Use bisection application to explore its behavior with the 5 functions
  - Choose different starting values (brackets)
  - The app does not check whether there is a zero in the bracket, so you can see what goes wrong...
  - Record your results; note interesting or odd behaviors

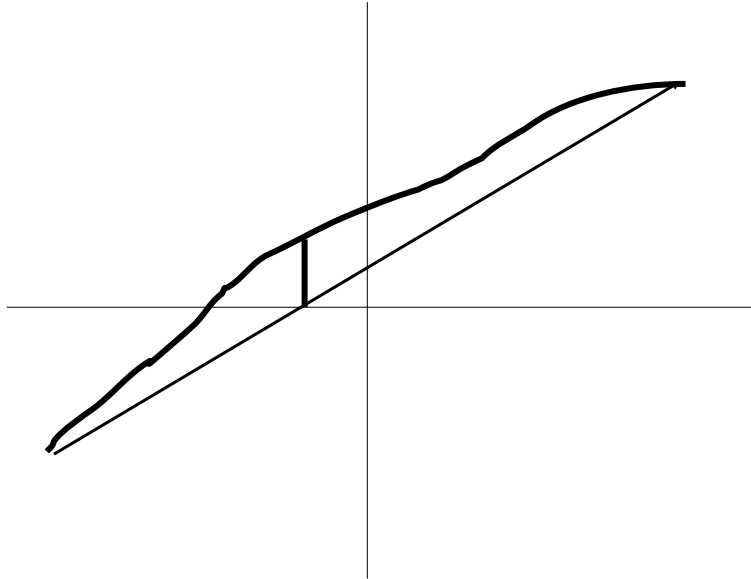
## Secant, False Position Methods

- For smooth functions:
  - Approximate function by straight line
  - Estimate root at intersection of line with axis
- Secant method:
  - Uses most recent 2 points for next approximation line
  - Faster than false position but doesn't keep root bracketed and may diverge
- False position method:
  - Uses most recent points that have opposite function values
- Brent's method is better than either and should be the only one you really use:
  - Combines bisection, root bracketing and quadratic rather than linear approximation
  - See p. 267 of Numerical Recipes in C

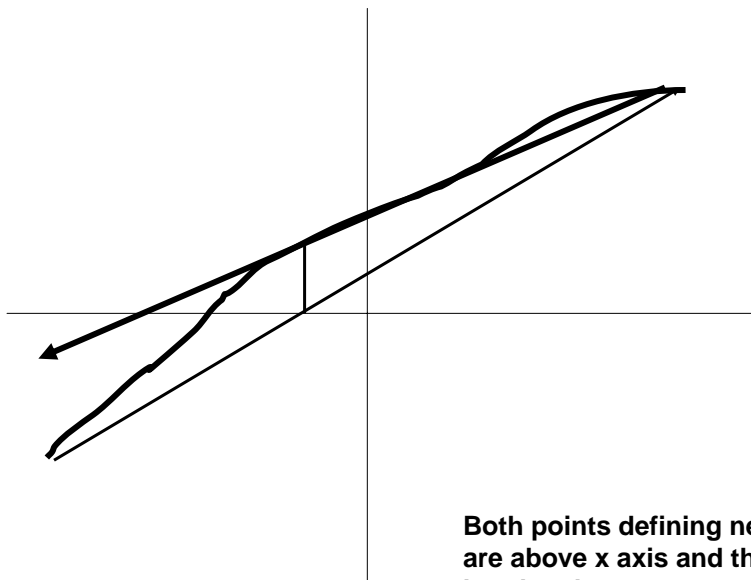
### Secant Method



## Secant Method

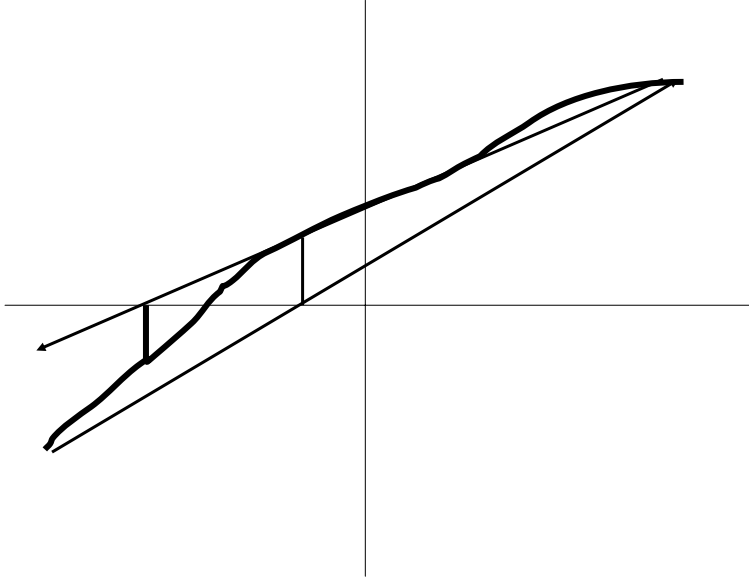


## Secant Method

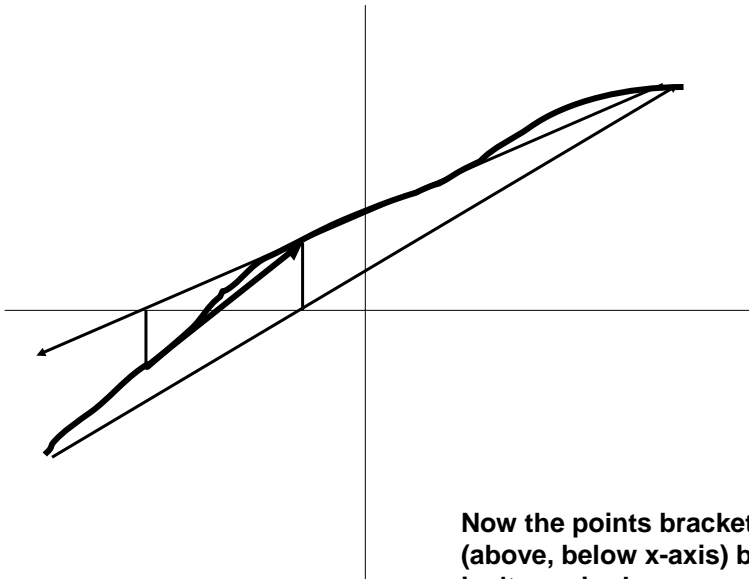


Both points defining new line are above x axis and thus don't bracket the root

## Secant Method

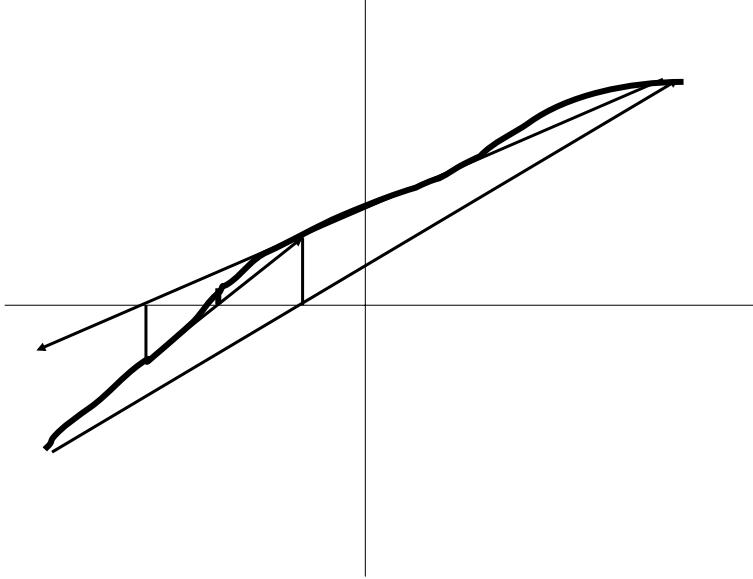


## Secant Method

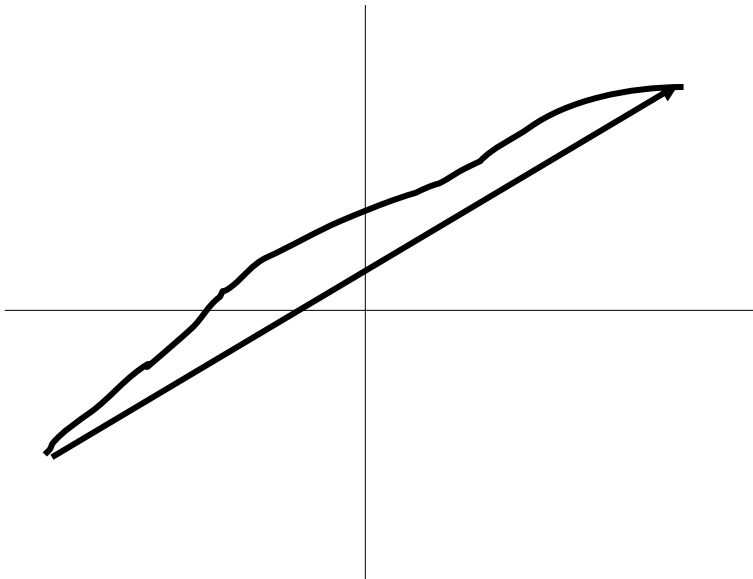


Now the points bracket the root  
(above, below x-axis) but this  
isn't required

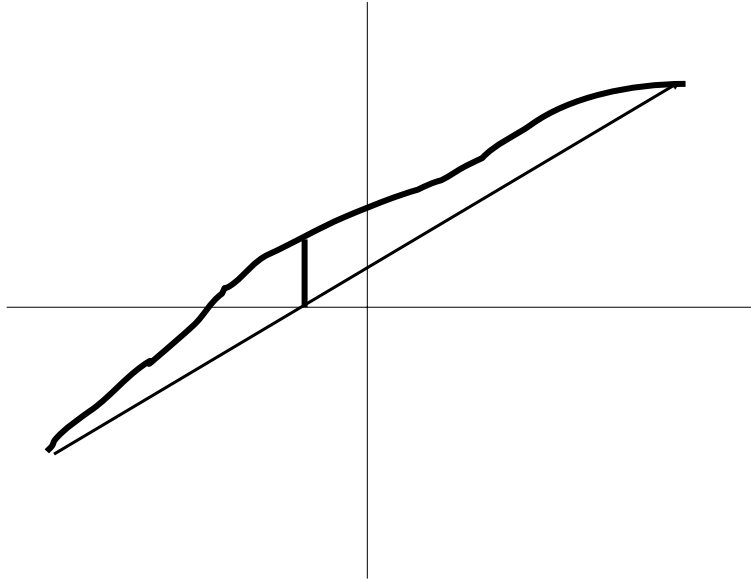
## Secant Method



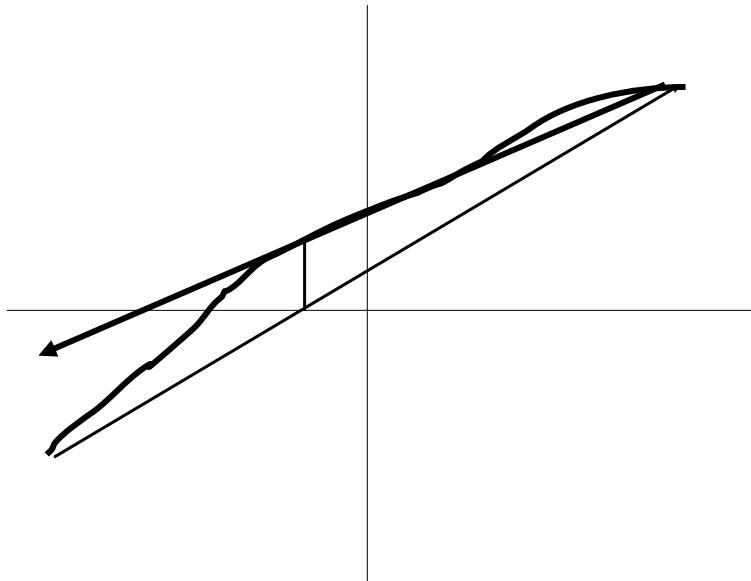
## False Position Method



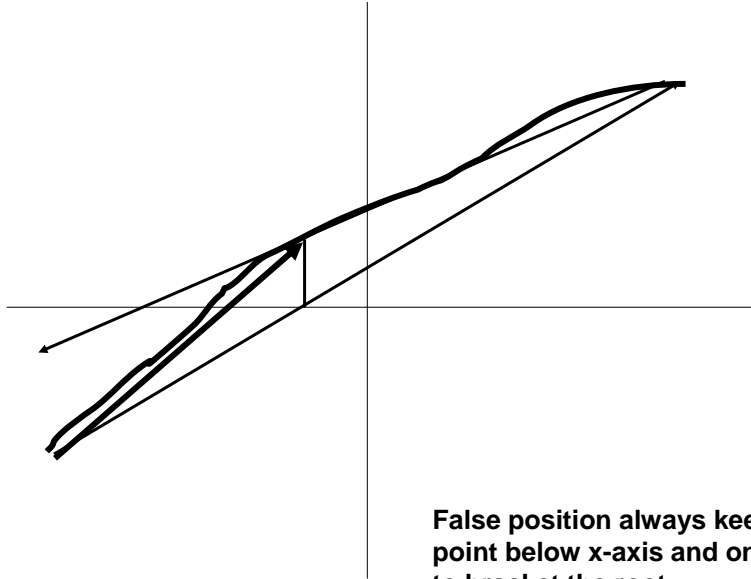
## False Position Method



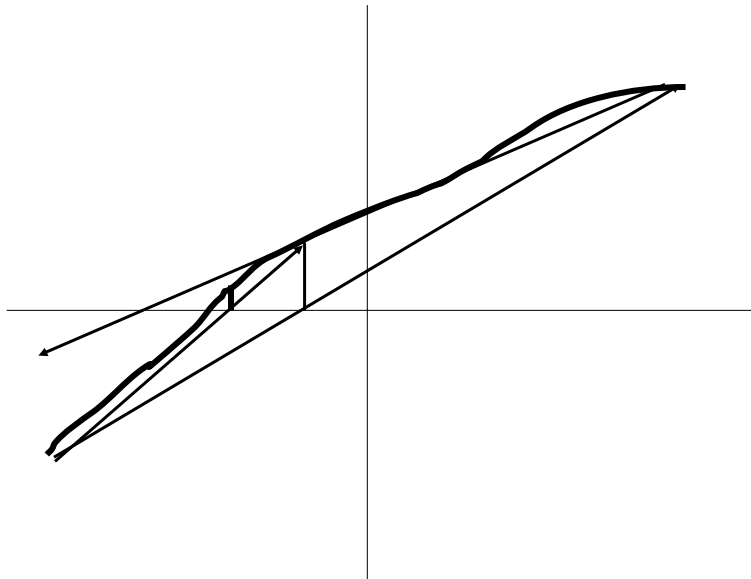
## False Position Method



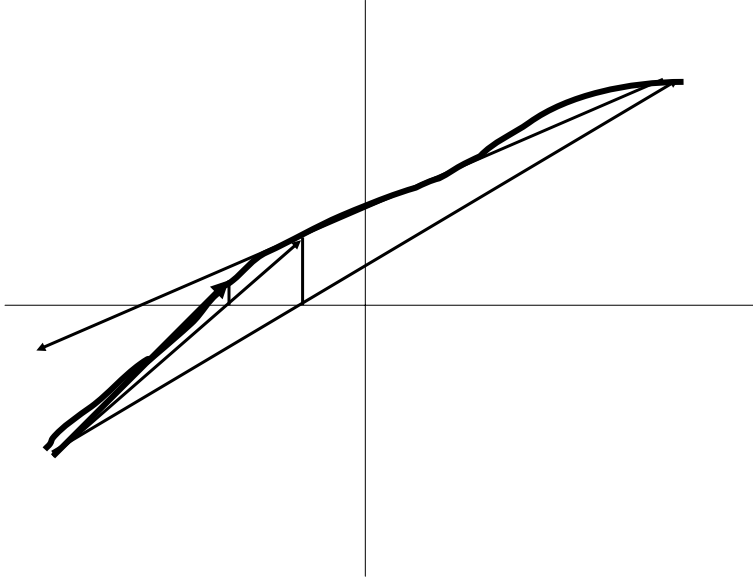
## False Position Method



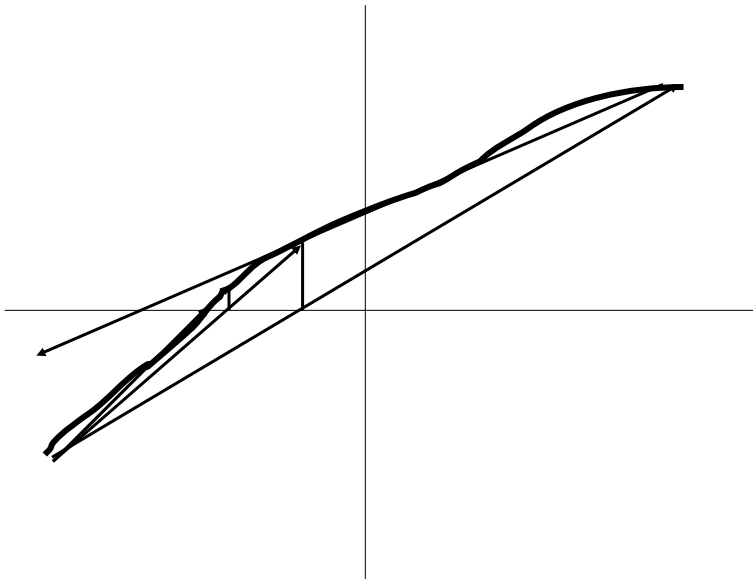
## False Position Method



## False Position Method



## False Position Method



## **Exercise**

- **Use secant method app to experiment with the 5 functions**
  - **Choose different starting values (brackets)**
  - **The app does not check whether there is a zero in the bracket, so you can see what goes wrong...**
  - **Record your results; note interesting or odd behaviors**