1.00 Lecture 25

Numerical Methods:
Root Finding

Reading for next time: Big Java: section 19.4

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

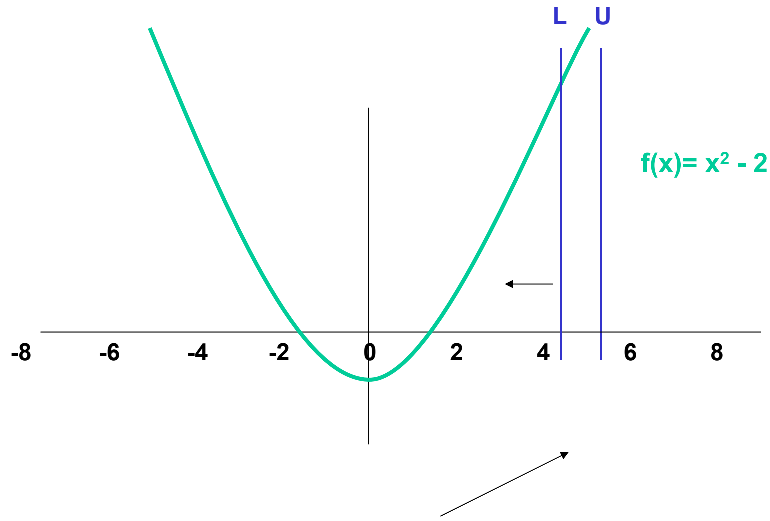
Root Finding Methods

- 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 Athena. Can translate or link to Java
  - The C code in the book is quite (needlessly) obscure
- Why is this so hard?
  - The computer can't "see" the functions. It only has function values at a few points. You'd find it hard to solve equations with this little information also!

Root Finding Preparation

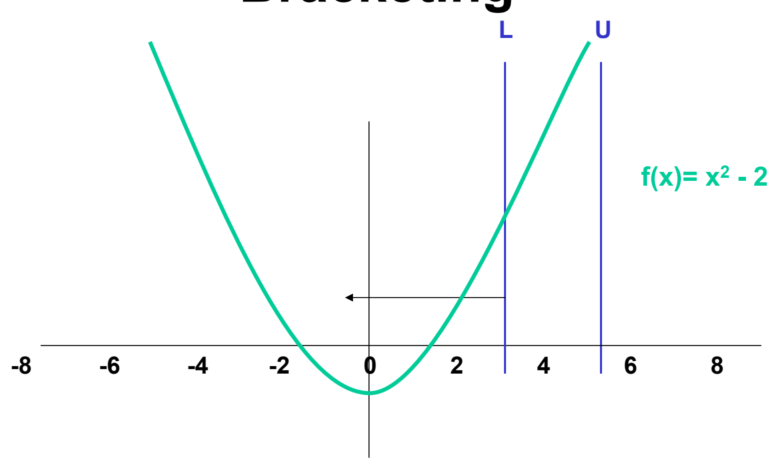
- Before using root finding methods:
  - Graph the equation(s): Matlab, etc.
    - Are they continuous, smooth; how differentiable?
  - Use Matlab, etc. to explore solutions
  - Linearize the equations and use matrix methods to get approximate solutions
  - Approximate the equations in other ways and solve analytically
  - Bracket the ranges where roots are expected
- For fun, look at $f(x) = 3x^2 - (1/\pi^4)\ln[(\pi - x)^2] + 1$
  - Plot it at 3.13, 3.14, 3.15, 3.16; f(x) is around 30
  - Well behaved except at x= π
  - Dips below 0 in interval x= π +/- 10^{-667}
  - This interval is less than precision of doubles!
    - You'll never find these two roots numerically
  - This is in Pathological.java: experiment with it later

Bracketing



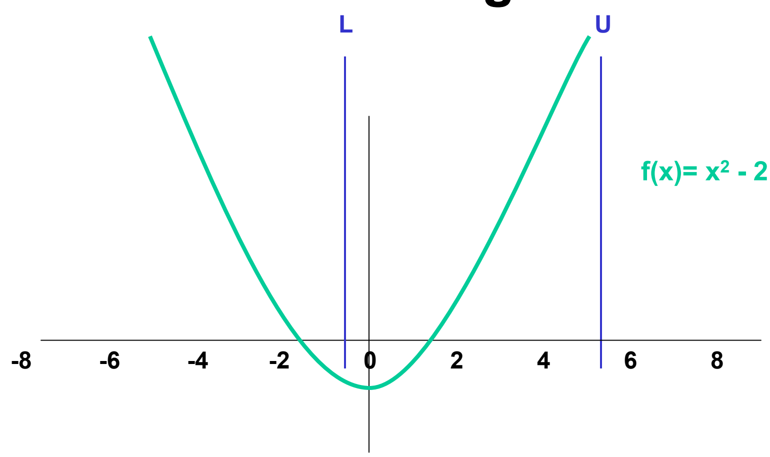
No zero in bracket (though we can't be sure)
Move in direction of smaller f(x) value.
Empirical multiplier of 1.6 to expand bracket size

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

"Function Passing" Again

```
// MathFunction is interface with one method
public interface MathFunction {
    public double f(double x);
}
```

```
// FuncA implements the interface
public class FuncA implements MathFunction {
    public double f(double x) {
        return x*x - 4;
    }
}
```

Bracketing Program

```
public class Bracket {
    public static boolean zbrac(MathFunction func, double[] x){
        // Java version of zbrac, p.352, 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
                                        // (Use JOption prompt)
        boolean intervalFound= zbrac(new FuncA(), bound);
        System.out.println("Bracket found? " + intervalFound);
        if (intervalFound)
            System.out.println(“L:"+bound[0]+" U: "+bound[1]);
        System.exit(0);
    }
}

// This program implements what the previous slide drawings show

// Numerical Recipes has 2nd bracketing program on p.352, which
// searches subintervals in bracket and records those w/zeros
```

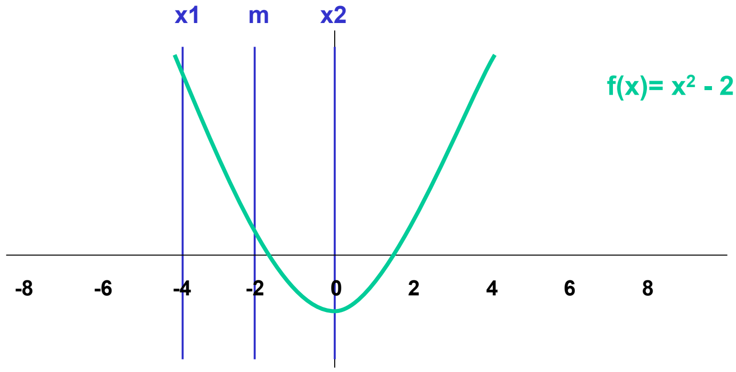
Paper Exercise: Brackets

- 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

Bisection

- Bisection
  - Interval passed as arguments 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 a 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(midpoint) == 0.0 (within some tolerance)
    - Has limit on number of iterations, etc.

Bisection

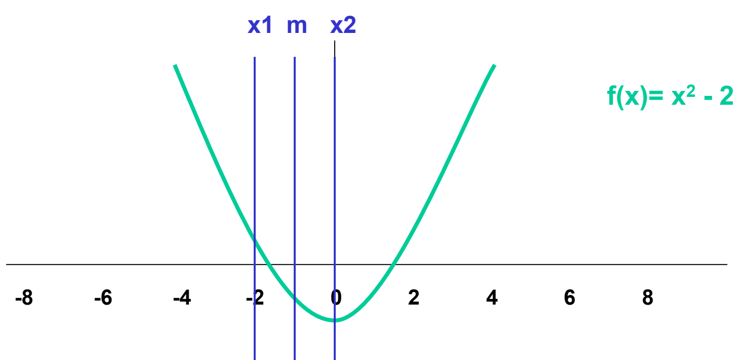


f(x1)*f(m) > 0, so no root in [x1, m]

f(m)*f(x2) < 0, so root in [m, x2]. Set x1=m

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

Bisection



f(m)*f(x2) > 0, so no root in [m, x2]

f(x1)*f(m) < 0, so root in [x1, m]. Set x2= m

Continue until (x2-x1) is small enough

Bisection- Simple Version

```
public class BisectSimple {
    public static double bisect(MathFunction func, double x1,
                double x2, double epsilon) {
        double m;
        // Very rare case of double loop variables being ok
        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;
    }

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

Bisection- NumRec Version

```
public class RootFinder {                    // NumRec, p. 354
    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 RootFinder.rtbis()

    // 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 (we don't here)
```

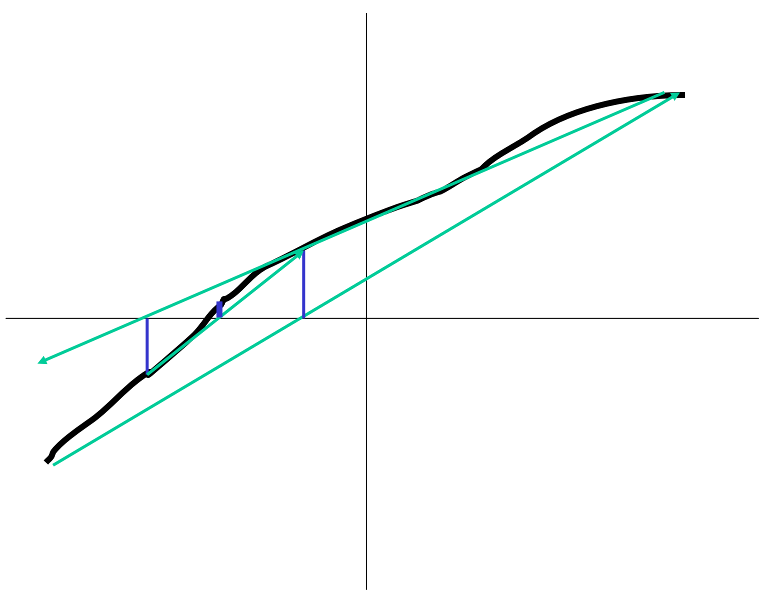
Exercise: Bisection

- Download Roots
- Use the bisection application in Roots to explore its behavior with the 5 functions
  - Choose different starting values (brackets) by clicking at two points along the x axis; red lines appear
  - Then just click anywhere. Each time you click, bisection will divide the interval; a yellow line shows the middle
  - When it thinks it has a root, the midline/dot turns green
  - 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

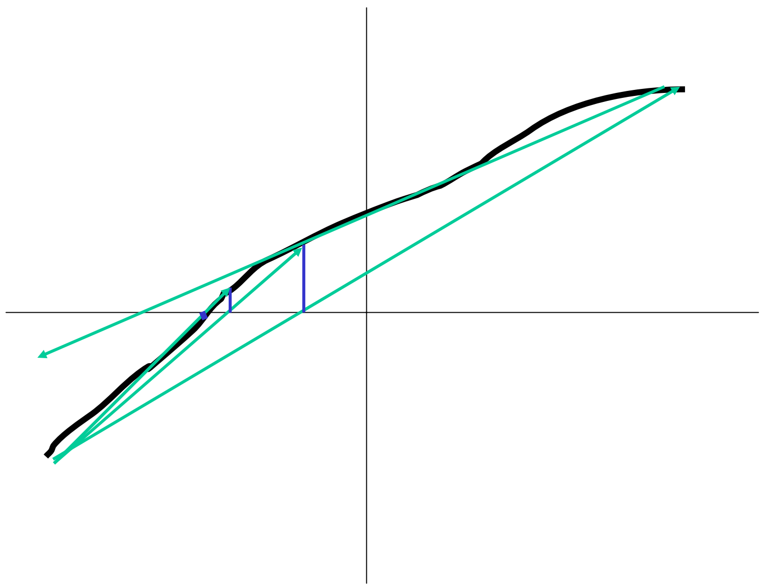
Secant, False Position Methods

- For smooth functions:
  - Approximate function by straight line
  - Estimate root at intersection of line with x 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. 360 of Numerical Recipes

Secant Method



False Position Method



Exercise

- Use secant method application in Roots to experiment with the 5 functions
  - Choose different starting values by clicking at two points along the x axis; red and orange lines appear
  - Then just click anywhere. When you click, a yellow secant line displays
  - Click again, and the intersection of secant and x axis is found, and the right and left lines (red and orange lines) move
  - When it thinks it has a root, the midline/dot turns green
  - The app does not check whether there is a zero in the limits, so you can see what goes wrong…
  - Record your results; note interesting or odd behaviors

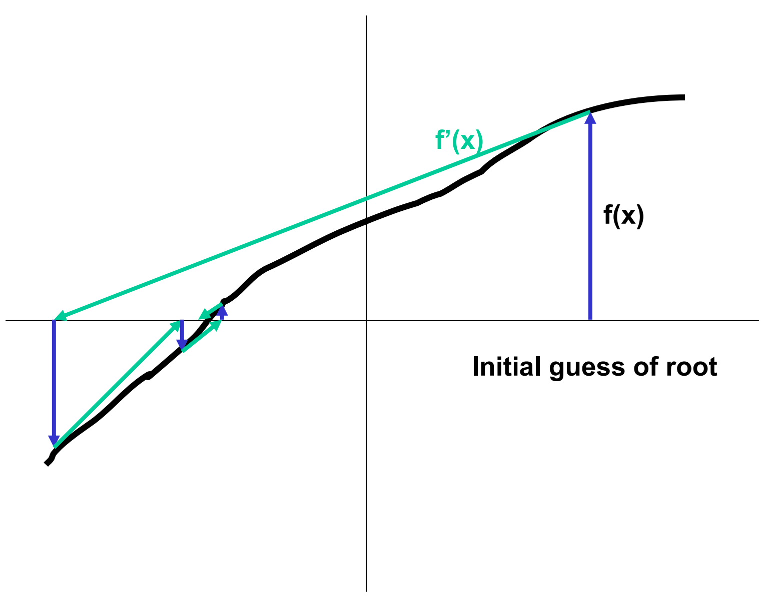
Newton's Method

- **Based on Taylor series expansion:**

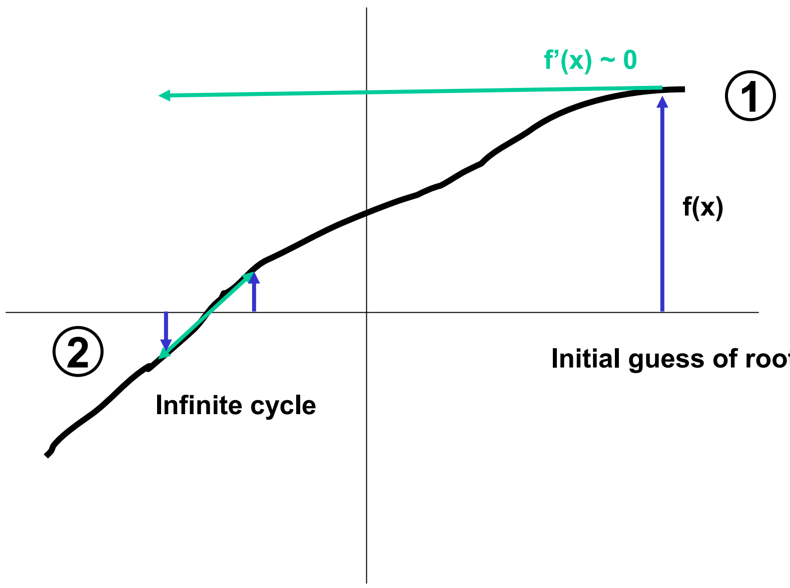
$$f(x+\delta) \approx f(x) + f'(x)\delta + f''(x)\delta^2/2 + \ldots$$

  - **For small increment and smooth function, higher order derivatives are small and** $f(x+\delta)=0$ **implies** $\delta = -f(x)/f'(x)$
  - **If high order derivatives are large or first derivative is small, Newton can fail miserably**
  - **Converges quickly if assumptions met**
  - **Has generalization to n dimensions that is one of the few available**
  - **See Numerical Recipes for 'safe' Newton-Raphson method, which uses bisection when first derivative is small, etc.**

Newton's Method



Newton's Method Pathologies



Newton's Method

```
public class Newton {                   // NumRec, p. 365
    public static double newt(MathFunction2 func, double a,
                                  double b, double epsilon) {
        double guess= 0.5*(a + b);  // No real bracket, only guess
        for (int j= 0; j < JMAX; j++) {
            double fval= func.fn(guess);
            double fder= func.fd(guess);
            double dx= fval/fder;
            guess -= dx;
            System.out.println(guess);
            if ((a - guess)*(guess - b) < 0.0) {
                System.out.println("Error: out of bracket");
                return ERR_VAL;           // Experiment with this
            }                             // It's conservative
            if (Math.abs(dx) < epsilon)
                return guess;
        }
        System.out.println("Maximum iterations exceeded");
        return guess;
    }
```

Newton's Method, p.2

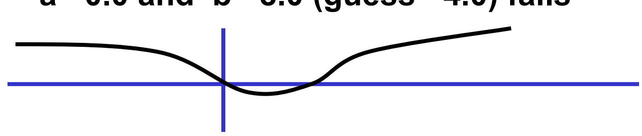
```
    public static int JMAX= 50;
    public static double ERR_VAL= -10E10;

    public static void main(String[] args) {
        double root= Newton.newt(new FuncB(), -0.0, 8.0, 0.0001);
        System.out.println("Root: " + root);
    }
}       // End Newton
```

```
public class FuncB implements MathFunction2 {
    public double fn(double x) {
        return x*x - 2;
    }
    public double fd(double x) {
        return 2*x;    }   }
```

```
public interface MathFunction2 {
    public double fn(double x);      // Function value
    public double fd(double x);   }  // 1st derivative value
```

Examples

- $f(x)= x^2 + 1$
  - No real roots, Newton generates 'random' guesses
- $f(x)= \sin(5x) + x^2 - 3$ Root= -0.36667
  - Try a= –1 and b = 2 (guess= 0.5)initially
  - Using a= 0 and b = 2 (guess= 1) will fail with conservative Newton (outside bracket)
- $f(x)= \ln(x^2 - 0.8x + 1)$ Roots= 0, 0.8
  - a= 0 and b =1.2 (guess= 0.6) works
  - a= 0.0 and b= 8.0 (guess= 4.0) fails



Exercise A

- Download Newton:
  - The functions on previous slide are implemented as FuncB, FuncC and FuncD
  - Newton takes doubles a and b as arguments, but they are not a bracket. It averages them to create its first guess
  - Experiment with different initial guesses
  - Solutions are on previous slide

Exercise B

- Use Newton's method application in Roots to experiment with the 5 functions
  - Choose starting guess by clicking at one point along the x axis; red line appears
  - Then just click anywhere. When you click, a yellow tangent line displays
  - Click again, and the intersection of tangent and x axis is found, and the guess (red line) moves
  - When it thinks it has a root, the line/dot turns green
  - The app does not check whether there is a zero in the limits, so you can see what goes wrong…
  - Record your results; note interesting or odd behaviors

Who Finds A Root?

Function	Bisection	Secant	Newton
3 sin(x)	Maybe	Maybe	Usually
$0.1x^2$	No	Yes	Yes
1/x	Yes	No	No
5 sin(x) / x	Maybe	Maybe	Maybe
sin (1/x)	Usually	Maybe	Maybe

Moral: You need to understand your function, its range, its likely zeros and the method you propose to use