1.00 Lecture 33

Numerical Methods: Root Finding

No .java files to upload in today's class; create a text file or .java file with roots tool results and upload it as your active learning solution

Reading for next time: Big Java 14.1-14.3

Root Finding in Nonlinear Equations

- **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. E.g.,**
    - **$x_0x_1^2 + 3x_1 = 20$**
    - **$x_0^3 – x_1^2 + x_0x_1 = 5$**
- **We address only the 1-D function**
  - **In 1-D, we can bracket the root between bounding values**
  - **In multidimensional case, it's impossible to bracket the root (as we see on the next slide)**
- **(Almost) all root finding methods are iterative**
  - **Start from an initial guess**
  - **Improve solution until convergence limit satisfied**
  - **Only smooth 1-D functions have convergence assured**

Solve f(x,y)=0 and g(x,y)=0

Image removed due to copyright restrictions. Figure 9.6.1 Solution of two nonlinear equations in two unknowns. From Press, William, Saul Teukolsky, et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992. See: http://www.nrbook.com/a/bookcpdf.php.

If n>2, find intersection of n unrelated zero contour hypersurfaces of dimension n-1

From Numerical Recipes

Root Finding Methods

- **Elementary (pedagogical use only):**
  - **Bisection**
  - **Secant**
- **"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 onAthena is MIT's UNIX-based computing environment. OCW does not provide access to it. Can translate or link to Java**
- **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. Exercise (0 to 20).**

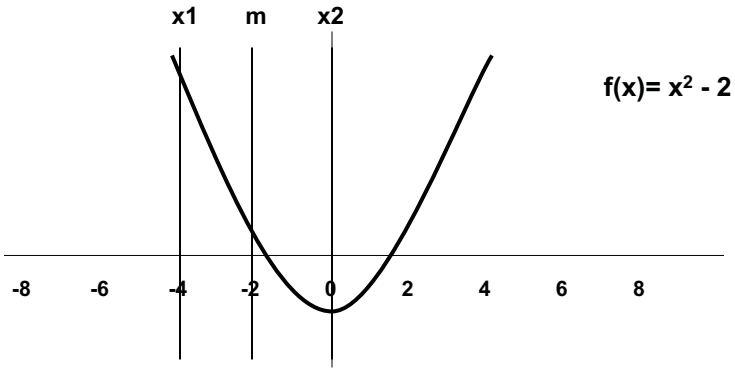
Root Finding Preparation

- Before using root finding methods:
  - Try to solve the equation(s) analytically. May be possible
    - Use Mathematica, etc. to check for analytical solutions
  - Graph the equation(s): Matlab, etc.
    - Are they continuous, smooth; how differentiable?
  - 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

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 two or more roots, bisection finds one
    - 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: scale
  - Numerical Recipes, p.354 gives the basic 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

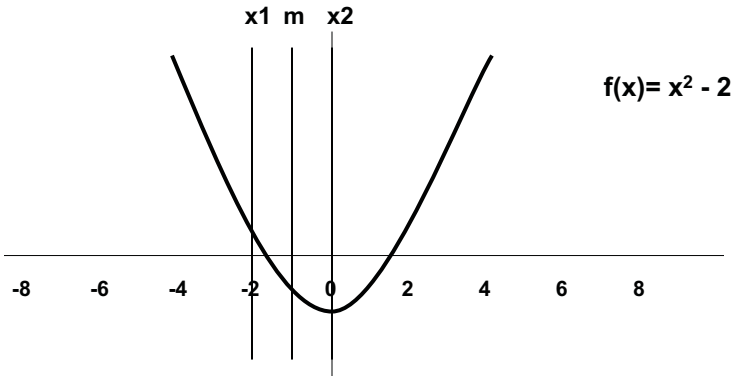


f(x1)*f(m) > 0, so no root in [x1, m]

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

"Function Passing" Again

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

```
// Quadratic implements the interface
public class Quadratic implements MathFunction {
    public double f(double x) {
        return x*x - 2;
    }
}
```

Bisection- Simple Version

```
public class BisectSimple {
  public static double bisect(MathFunction func, double x1,
                double x2, double epsilon) {
        double m;
        // 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 Quadratic(), -1.0, 8.0, 1E-15);
   System.out.println("Root: " + root);
   System.out.println("Sqrt:  "+ -Math.sqrt(2.0));  // As a check
  }
}
```

Bisection- NumRec Version

```
public class RootFinder {                    // NumRec, p. 354
    public static final int JMAX= 100;       // 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 can be 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)
    // Can use as 'black box', like classes in java.util, etc.
```

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 magenta 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
  - "Roots" is persnickety:
    - It throws away any segment with f*f >=0. ~~It looks at both sides.~~

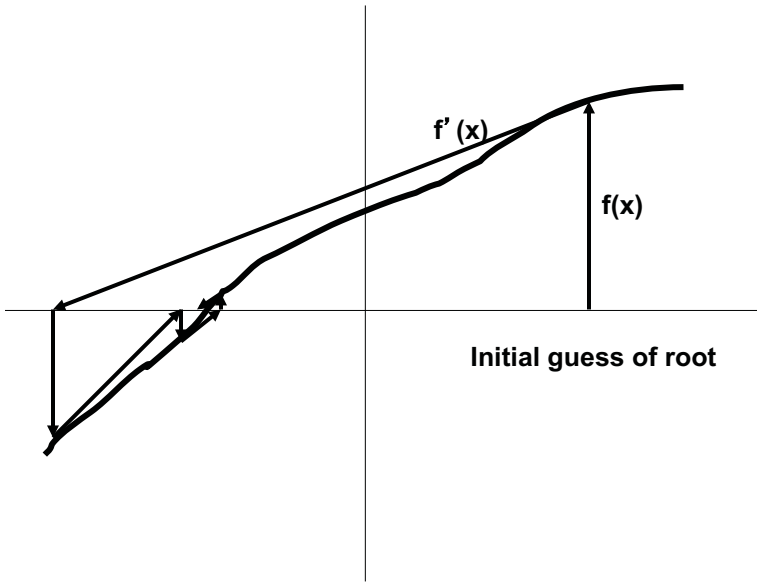
Newton's Method

- Based on Taylor series expansion:

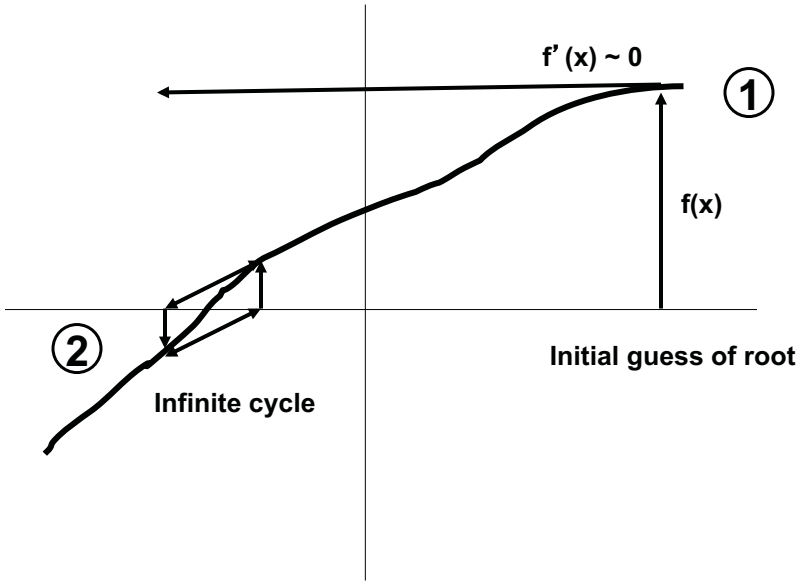
$$f(x+\delta) \approx f(x) + f'(x)\delta + f''(x)\delta^2/2 + \ldots$$

  - 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.
    - rtsafe, page 366; Java version in your download

Newton's Method



Newton's Method Pathologies



Newton's Method

```
public class Newton {                // NumRec, p. 365
    public static double newt(MathFunctionNewton 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;      // 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= 100;
    public static double ERR_VAL= -10E10;

    public static void main(String[] args) {
        double root= Newton.newt(new Quad(), -1.0, 8.0, 1E-15);
        System.out.println("Root: " + root);
    }
}       // End Newton
```

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

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

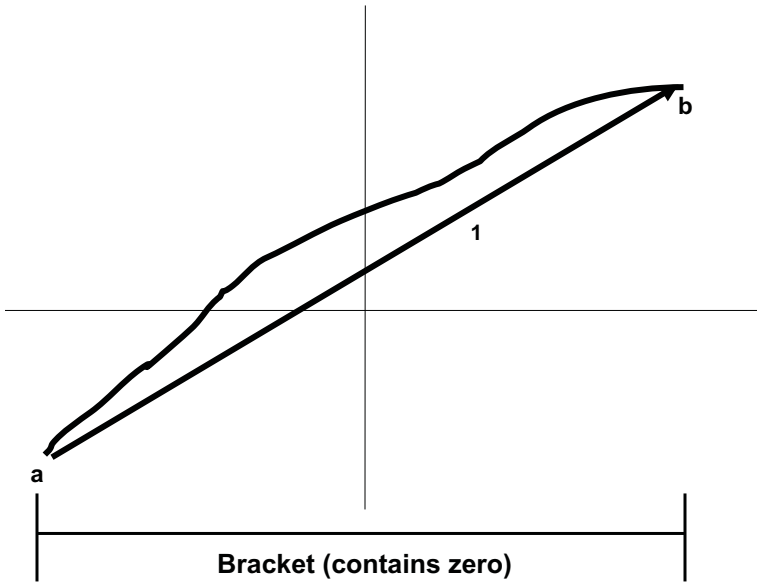
Exercise

- 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 magenta 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

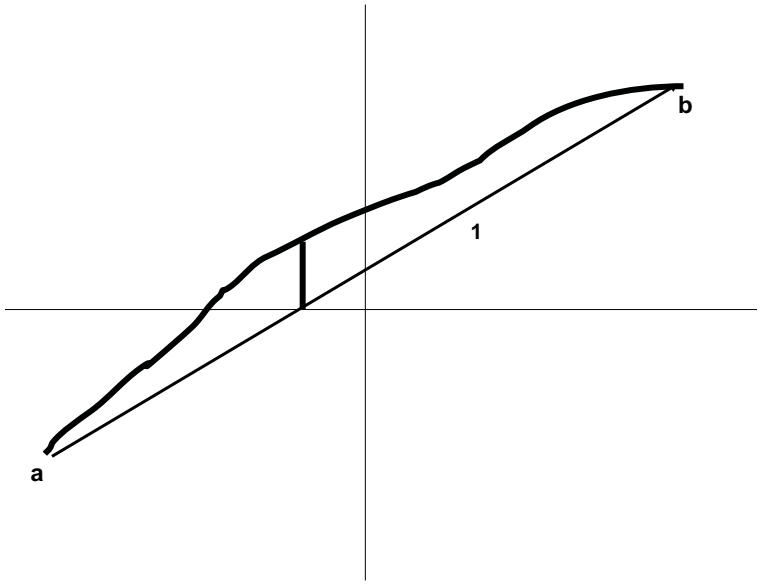
Secant Method

- 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
  - Does not keep root bracketed
  - "False position" variation keeps root bracketed, but is slower
- Brent's method is better than secant 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. Java version is in your download.

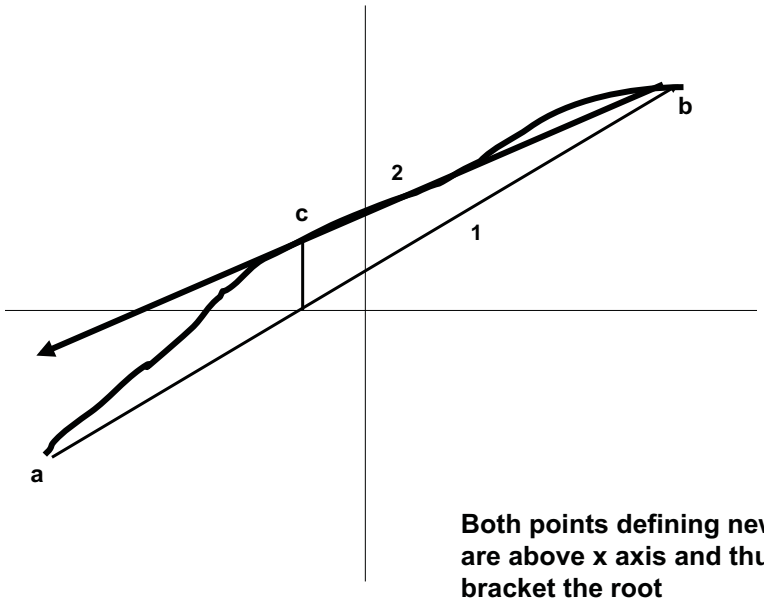
Secant Method



Secant Method

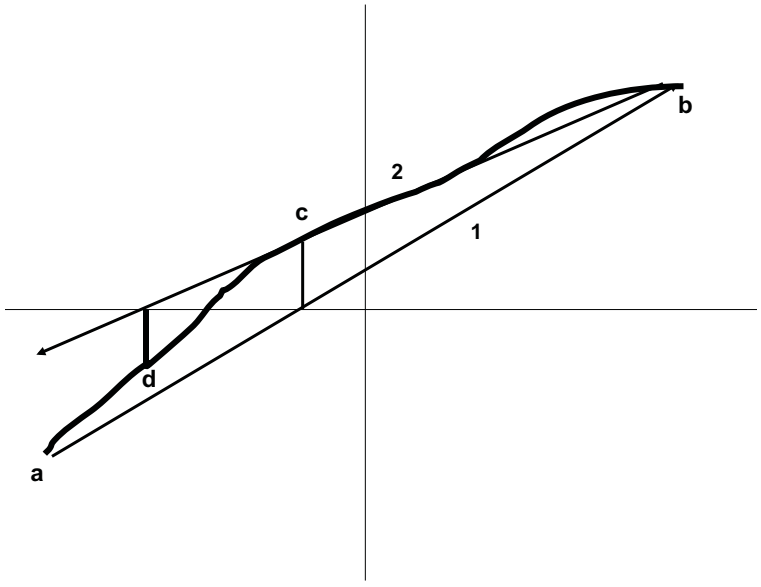


Secant Method

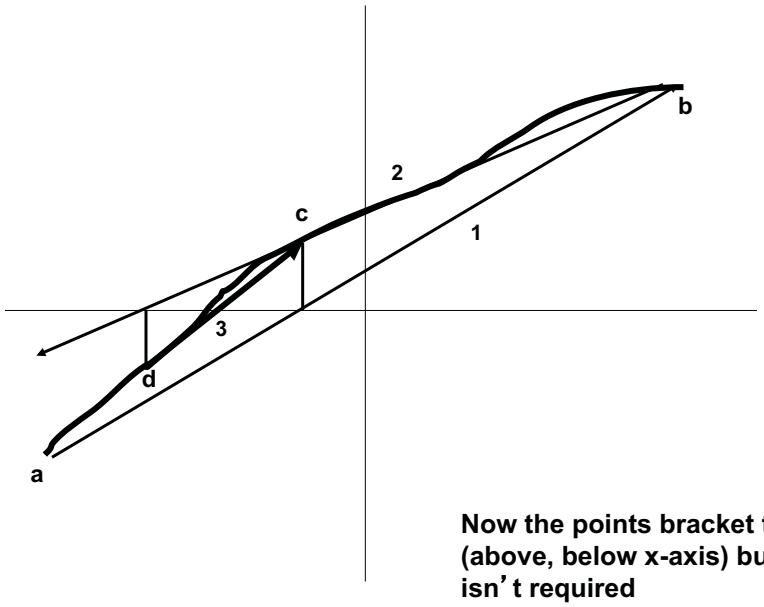


Both points defining new line are above x axis and thus don't bracket the root

Secant Method

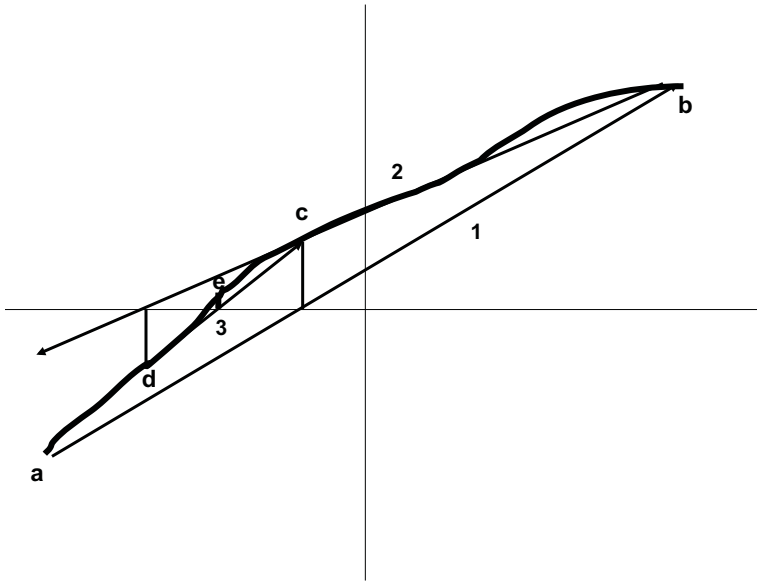


Secant Method



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

Secant Method



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 magenta 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**

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012