

1.00 Lecture 20

October 27, 2005

More on root finding
Numerical Integration

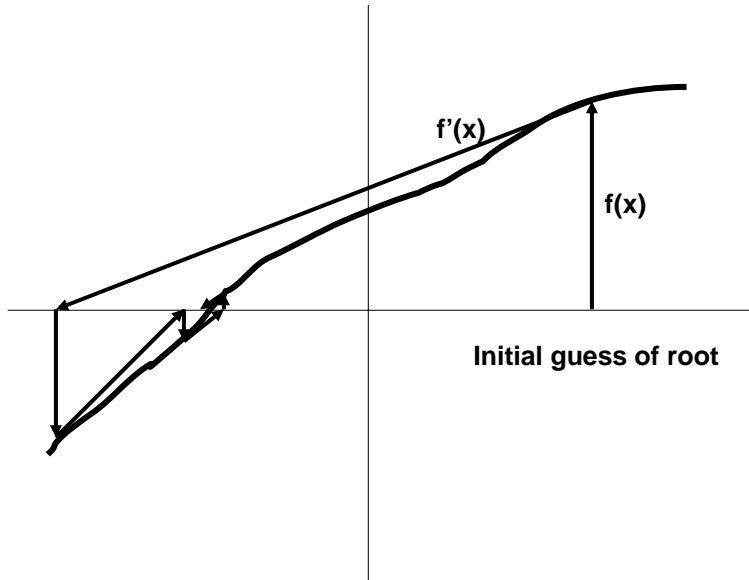
Newton's Method

- **Based on Taylor series expansion:**

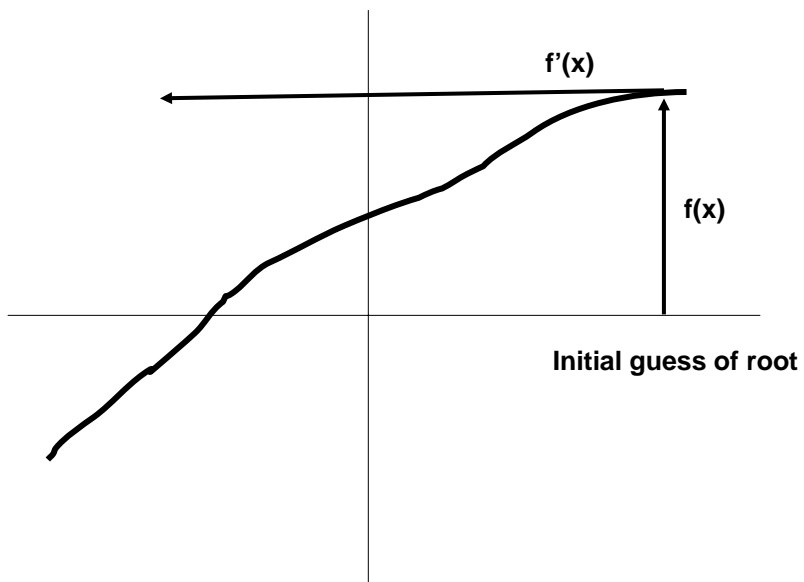
$$f(x+\delta) \approx f(x) + f'(x)\delta + f''(x)\delta^2/2 + \dots$$

- 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 1st 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 1st derivative is small, etc.

Newton's Method



Newton's Method Pathology



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);
        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);
    System.exit(0);
}

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
 - Bracket between -1 and 2, for example
 - Bracket between 0 and 2 will fail with conservative Newton (outside bracket)
- $f(x) = \ln(x^2 - 0.8x + 1)$ Roots = 0, 0.8
 - Bracket between 0 and 1.2 works
 - Bracket between 0.0 and 8.0 fails

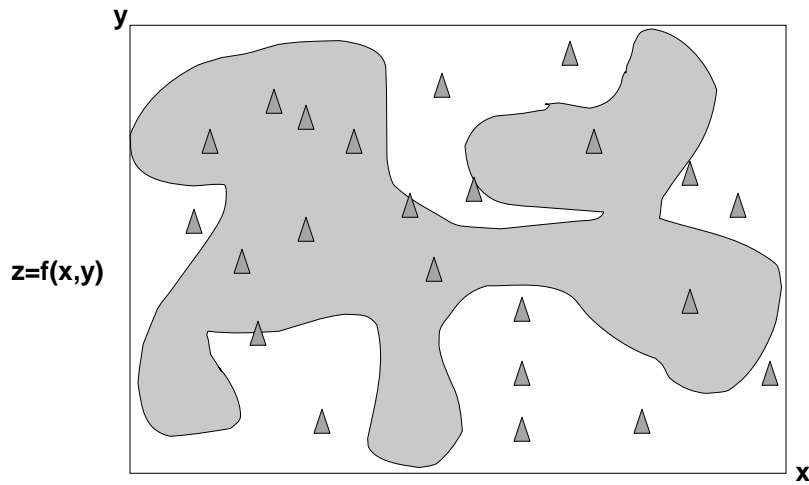


- Use Roots.java from lab to experiment!

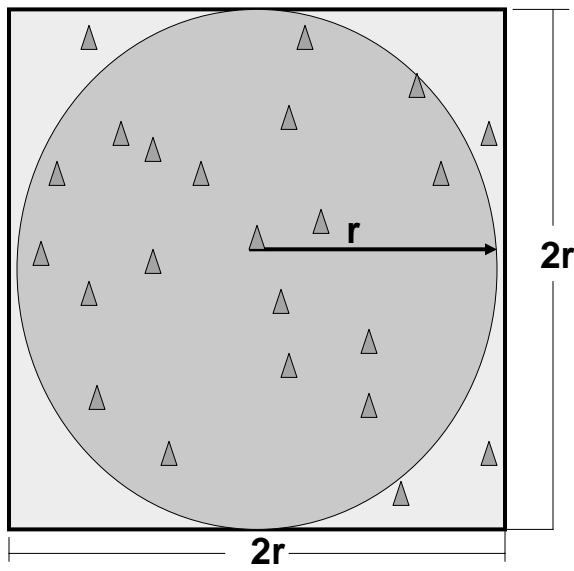
Numerical Integration

- Classical methods are of historic interest only
 - Rectangular, trapezoid, Simpson's
 - Work well for integrals that are very smooth or can be computed analytically anyway
- Extended Simpson's method is only elementary method of some utility for 1-D integration
- Multidimensional integration is tough
 - If region of integration is complex but function values are smooth, use Monte Carlo integration
 - If region is simple but function is irregular, split integration into regions based on known sites of irregularity
 - If region is complex and function is irregular, or if sites of function irregularity are unknown, give up
- We'll cover 1-D extended Simpson's method only
 - See Numerical Recipes chapter 4 for more

Monte Carlo Integration



Finding Pi



Randomly generate points in square $4r^2$. Odds that they're in the circle are $\Pi r^2 / 4r^2$, or $\Pi / 4$.

This is Monte Carlo Integration, with $f(x,y) = 1$

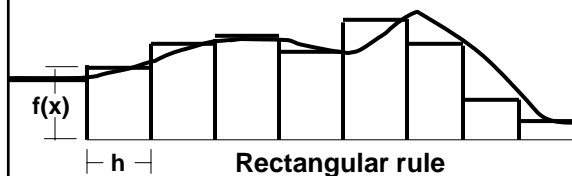
If $f(x,y)$ varies slowly, then evaluate $f(x,y)$ at each sample point in limits of integration and sum

FindPi

```
public class FindPi {
    public static double getPi () {
        int count=0;
        for (int i=0; i < 1000000; i++) {
            double x= Math.random() - 0.5; // Ctr at 0,0
            double y= Math.random() - 0.5;
            if ((x*x + y*y) < 0.25) // If in region
                count++; // Increment integral
        } // More generally, eval f()
        return 4.0*count/1000000.0; // Integral value
    }

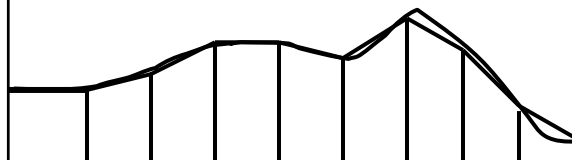
    public static void main(String[] args) {
        System.out.println(getPi ());
        System.exit(0);
    }
}
```

Elementary Methods



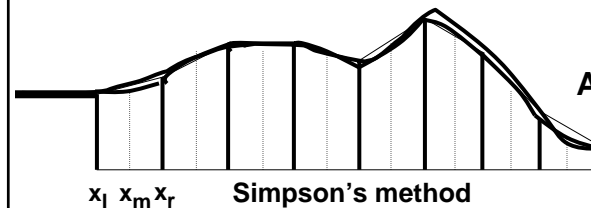
$$A = f(x_{\text{left}}) * h$$

Rectangular rule



$$A = (f(x_{\text{left}}) + f(x_{\text{right}})) * h / 2$$

Trapezoidal rule



$$A = (f(x_l) + 4f(x_m) + f(x_r)) * h / 6$$

Simpson's method

Elementary Methods

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

public class Integration {
    public static double rect(MathFunction func,
        double a, double b, int n) {
        double h= (b-a)/n;
        double answer=0.0;
        for (int i=0; i < n; i++)
            answer += func.f(a+i*h);
        return h*answer;    }

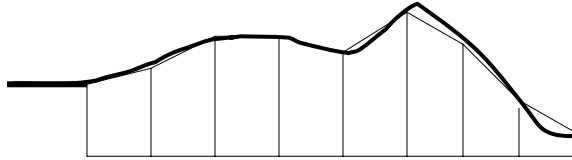
    public static double trap(MathFunction func,
        double a, double b, int n) {
        double h= (b-a)/n;
        double answer= func.f(a)/2.0;
        for (int i=1; i <= n; i++)
            answer += func.f(a+i*h);
        answer -= func.f(b)/2.0;
        return h*answer;    }
```

Elementary Methods, p.2

```
public static double simp(MathFunction func,
    double a, double b, int n) {
    // Each panel has area (h/6)*(f(x) + 4f(x+h/2) + f(x+h))
    double h= (b-a)/n;
    double answer= func.f(a);
    for (int i=1; i <= n; i++)
        answer += 4.0*func.f(a+i*h-h/2.0)+ 2.0*func.f(a+i*h);
    answer -= func.f(b);
    return h*answer/6.0;    }

public static void main(String[] args) {
    double r= Integration.rect(new FuncA(), 0.0, 8.0, 200);
    System.out.println("Rectangle: " + r);
    double t= Integration.trap(new FuncA(), 0.0, 8.0, 200);
    System.out.println("Trapezoid: " + t);
    double s= Integration.simp(new FuncA(), 0.0, 8.0, 200);
    System.out.println("Simpson: " + s);
    System.exit(0);
}
} //Problems: no accuracy estimate, inefficient, only closed int
```

Better Trapezoid Rule



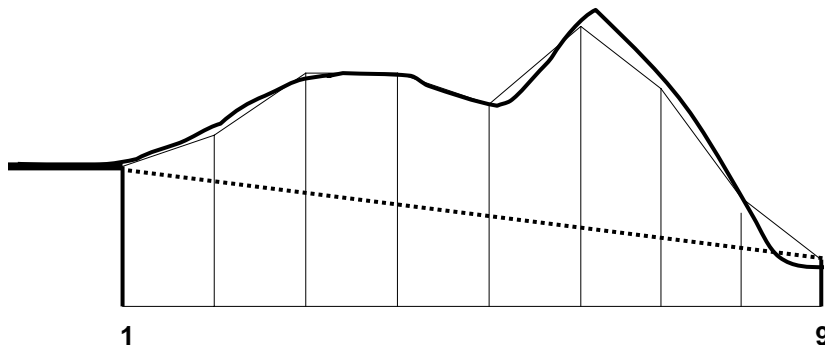
Individual trapezoid approximation:

$$\int_{x_1}^{x_2} f(x) dx = h(0.5 f_1 + 0.5 f_2) + O(h^3 f''')$$

Use this N-1 times for (x_1, x_2) , (x_2, x_3) , ..., (x_{N-1}, x_N) and add the results:

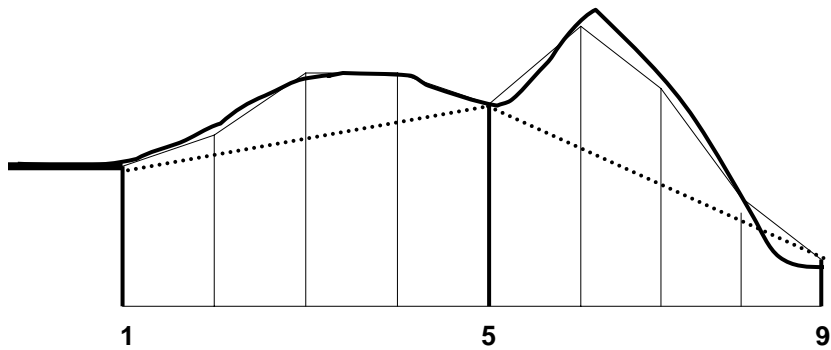
$$\int_{x_1}^{x_N} f(x) dx = h(0.5 f_1 + f_2 + \dots + f_{N-1} + 0.5 f_N) + O((b-a)^3 f''' / N^2)$$

Better Trapezoid Rule



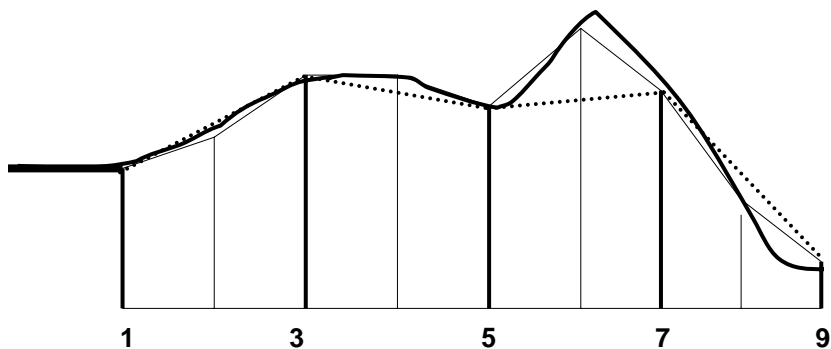
N=1, requires two function evaluations

Better Trapezoid Rule



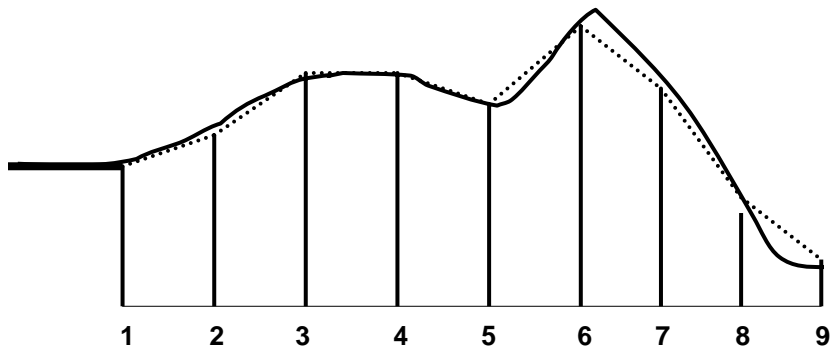
$N=2$, requires only one more function evaluation

Better Trapezoid Rule



$N=4$, requires only two more function evaluations

Better Trapezoid Rule



N=8, requires only 4 more function evaluations

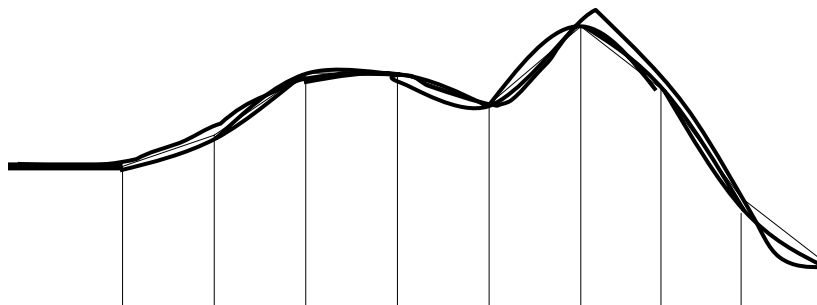
Using Trapezoidal Rule

- **Keep cutting intervals in half until desired accuracy met**
 - Estimate accuracy by change from previous estimate
 - Each halving requires relatively little work because past work is retained
- **By using a quadratic interpolation (Simpson's rule) to function values instead of linear (trapezoidal rule), we get better error behavior**
 - By good fortune, errors cancel well with quadratic approximation used in Simpson's rule
 - Computation same as trapezoid, but uses different weighting for function values in sum

Extended Trapezoid Method

```
class Trapezoid { // NumRec p. 137
public static double trapzd(MathFunction func, double a,
                           double b, int n) {
    if (n==1) {
        s= 0.5*(b-a)*(func.f(a)+func.f(b));
        return s; }
    else {
        int it= 1; // Addl interior points
        for (int j= 0; j < n-2; j++)
            it *= 2; // Subdivisions
        double tnm= it; // Double value of it
        double del ta= (b-a)/tnm; // Spacing of points
        double x= a+0.5*del ta; // Pt to evaluate f(x)
        double sum= 0.0; // Contrib of new pts x
        for (int j= 0; j < it; j++) {
            sum += func.f(x);
            x+= del ta; }
        s= 0.5*(s+(b-a)*sum/tnm); // Value of integral
        return s; } }
private static double s; } // Current value of integral
```

Extended Simpson Method



Approximate function with quadratic, not linear form

Extended Simpson Method

```
public class Simpson { // NumRec p. 139
    public static double qsimp(MathFunction func, double a,
                               double b) {
        double ost= -1.0E30;
        double os= -1E30;
        for (int j=0; j < JMAX; j++) {
            double st= Trapezoid.trapzd(func, a, b, j+1);
            s= (4.0*st - ost)/3.0; // See NumRec eq. 4.2.4
            if (j > 4) // Avoid spurious early convergence
                if (Math.abs(s-os) < EPSILON*Math.abs(os) ||
                    (s==0.0 && os==0.0)) {
                    System.out.println("Simpson iter: " + j);
                    return s; }
            os= s;
            ost= st; }
        System.out.println("Too many steps in qsimp");
        return ERR_VAL; }
    private static double s;
    public static final double EPSILON= 1.0E-15;
    public static final int JMAX= 50;
    public static final double ERR_VAL= -1E10; }
```

Using the Methods

```
public static void main(String[] args) {
    // Simple example with just trapzd (see NumRec p. 137)
    System.out.println("Simple trapezoid use");
    int m= 20; // Want 2^m+1 steps
    int j= m+1;
    double ans= 0.0;
    for (j=0; j <=m; j++) { // Must use Trapzd in loop!
        ans= Trapezoid.trapzd(new FuncC(), 0.0, 8.0, j+1);
        System.out.println("Iteration: " + (j+1) + "
                           Integral: " + ans); }
    System.out.println("Integral: " + ans);
    // Example using extended Simpson method
    System.out.println("Simpson use");
    ans= qsimp(new FuncC(), 0.0, 8.0);
    System.out.println("Integral: " + ans);
    System.exit(0); } // End Simpson class

class FuncC implements MathFunction {
    public double f(double x) {
        return x*x*x*x + 2; } }
```

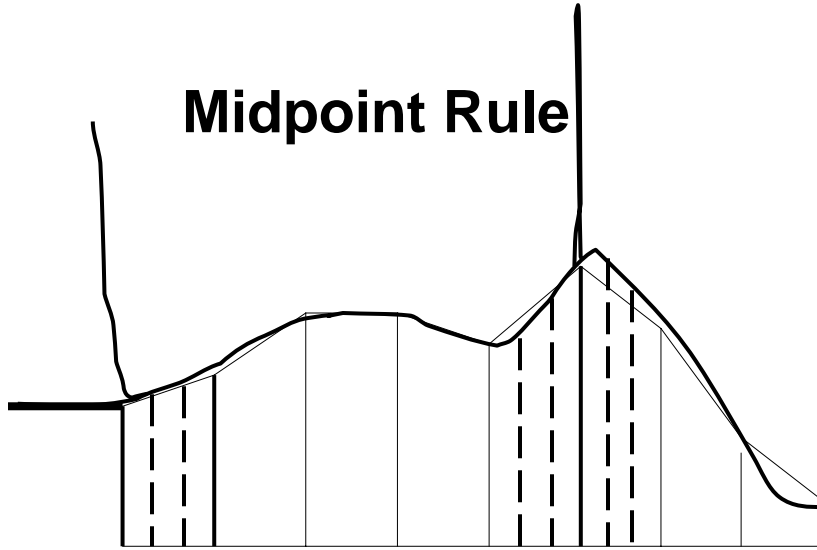
Romberg Integration

- **Generalization of Simpson (NumRec p. 140)**
 - Based on numerical analysis to remove more terms in error series associated with the numerical integral
 - Uses trapezoid as building block as does Simpson
 - Romberg is adequate for smooth (analytic) integrands, over intervals with no singularities, where endpoints are not singular
 - Romberg is much faster than Simpson or the elementary routines. For a sample integral:
 - Romberg: 32 iterations
 - Simpson: 256 iterations
 - Trapezoid: 8192 iterations

Improper Integrals

- **Improper integral defined as having integrable singularity or approaching infinity at limit of integration**
 - Use extended midpoint rule instead of trapezoid rule to avoid function evaluations at singularities or infinities
 - Must know where singularities or infinities are
 - Use change of variables: often replace x with $1/t$ to convert an infinity to a zero
 - Done implicitly in many routines
- **Last improvement: Gaussian quadrature**
 - In Simpson, Romberg, etc. the x values are evenly spaced. By relaxing this, we can get better efficiency and often better accuracy

Midpoint Rule



See Numerical Recipes for discussion, code

Adaptive Integration

```
public static double adapt(MathFunction func, double a,
    double b, int nPanels, double tol)
{
    double val, check;
    val = simp(func,a,b,2*nPanels);
    check = Math.abs(simp(func,a,b,nPanels)-val);
    if(check > tol)
        val = adapt(func,a,(a+b)/2.0,nPanels,tol) +
            adapt(func,(a+b)/2.0,b,nPanels,tol);
    return(val);
}
```