1.00 Lecture 24

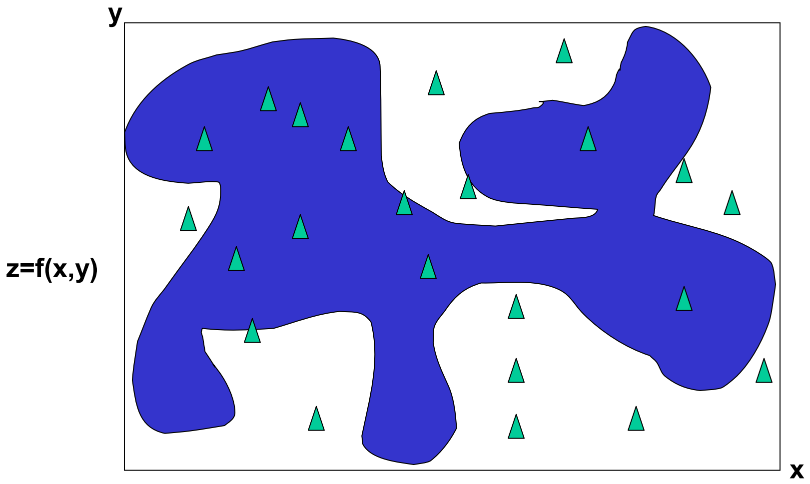
Integration

Reading for next time: Numerical Recipes, pp. 347-368
(Get as far as you're comfortable)

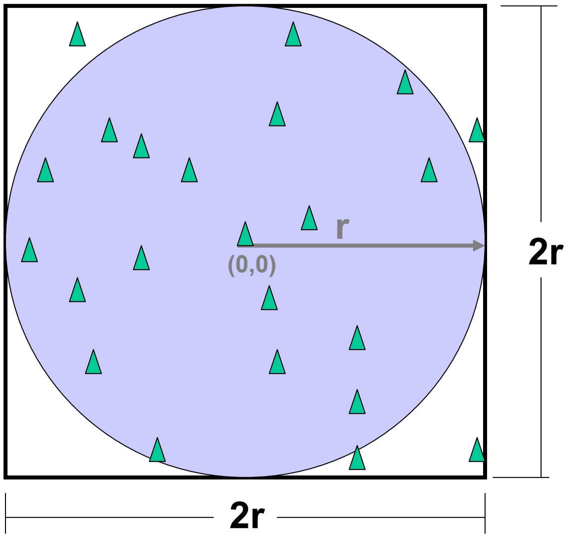
Numerical Integration

- 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 (first exercise)
  - 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



Integrate f(x,y) over Circular Area



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

Integration over Circular Area

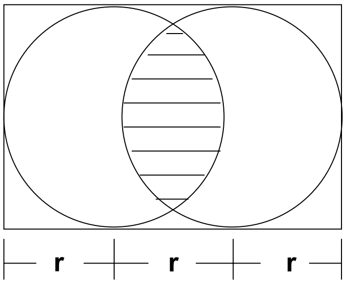
```
public class MonteCarloIntegration {
    public static double circularIntegral() {
        int nIter= 1000000;
        double sum= 0.0, radius= 0.5;
        for (int i=0; i < nIter; i++) {
            // Math.random() returns double d: 0 <= d <= 1
            double x= Math.random() - radius;  // Ctr at 0,0
            double y= Math.random() - radius;
            double f= 1.0;       // f(x,y)—constant here
            if ((x*x + y*y) < radius*radius) // If in region
                sum += f;        // Increment integral sum
        }
        return sum/nIter;  // Integral value
    }
    public static void main(String[] args) {
        System.out.println(“Result: “ +circularIntegral() );
        System.out.println(“Pi: “+ 4.0*circularIntegral() );
}   }
```

Integration over Circular Area, 2

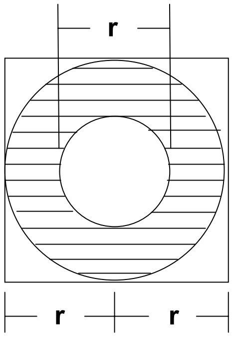
```
// To integrate f(x,y) = exp (x)/(y*y+1) over this area:

public class MonteCarloIntegration2 {
    public static double circularIntegral() {
       // for loop, random x, y same as previous slide
       // …
            double f= Math.exp(x)/(y*y+1);
            if ((x*x + y*y) < radius*radius) // If in region
                sum += f;        // Increment integral sum
        }
        return sum/nIter;  // Integral value
    }
    public static void main(String[] args) {
        System.out.println(“Result: “ +circularIntegral() );
    }
}
// Numerical integration is used when functions and areas
// of integration are really complex and ugly!
```

Exercises: Take Your Pick

- Find the shaded area within circles below:
  - Use circularIntegral() as your starting point
  - Use f(x,y)= 1 to find the areas below using integration
  - Equation of circle is $(x-x_c)^2 + (y-y_c)^2 = r^2$



(Answer is ~π/15, or .209)



(Answer is 3π/16, or .589)

Packaging Functions in Objects

- Consider writing a method that filters, integrates or finds the roots of a function:
  - f(x)= 0 on some interval [a, b], or find f(c)
- A general method that does this should have f(x) as an argument
  - Can't pass functions in Java (unlike C++)
  - Wrap the function in an object instead
    - Then pass the object reference to the filter, integration or root finding method as an argument
  - Define an interface that describes the object that will be passed to the numerical method
    - It must have a method, typically called f, that returns the value of the function f at a point defined by the arguments

Exercise: Passing Functions

- Write an interface MathFunction2 (New->Interface)

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

- Write a class FuncB that implements the interface for the function $5x_1^2 + 2x_2^3$ (New->Class)

```
public class FuncB implements MathFunction2 { … }
```

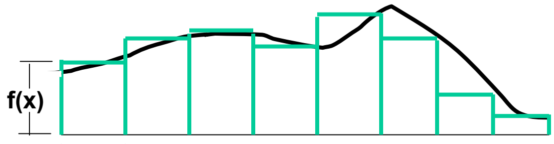
- Write a class Filter that contains a method filterFunc() that filters (postprocesses) functions: (New->Class)
  - filterFunc() takes a MathFunction2 object and two doubles d1 and d2 as arguments
  - It returns true if f(d1, d2) >= 0 and false otherwise (simple filter)

```
public class Filter {
  public static boolean filterFunc(MathFunction func,
                                   double d1, double d2){…}
```

- Write a main() method, in class Filter that:
  - Invokes filterFunc(), passing a FuncB object and two doubles x_1=2 and x_2=-3, and prints the boolean value returned

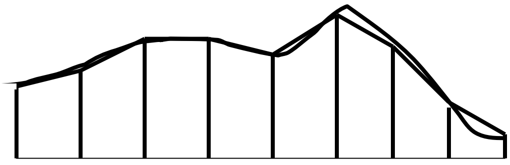
---

Elementary Integration Methods



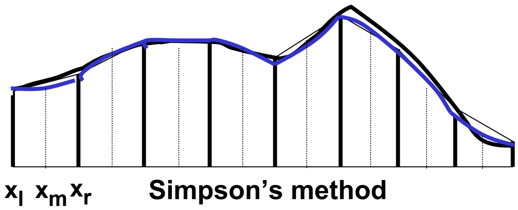
Rectangular rule

$A= f(x_{right})*h$



Trapezoidal rule

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



Simpson's method

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

Elementary Integration Methods

```
public 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;    }
```

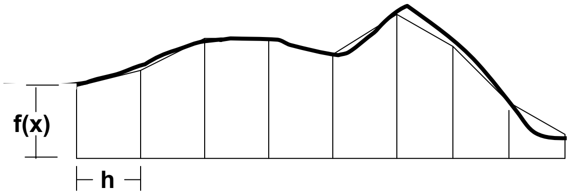
Elementary Integration Methods, p.2

```
 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
```

Exercise

- Download and run Integration
  - The function is $f(x)= x^4 + 2$
  - The integral is $\int_0^8 (x^4 + 2)dx = (x^5/5 + 2x)\Big|_0^8$
  - What value do rectangular, trapezoidal and Simpson give for the function provided?
  - Compute the correct value via calculus
  - Which is the most accurate?

Trapezoid Rule



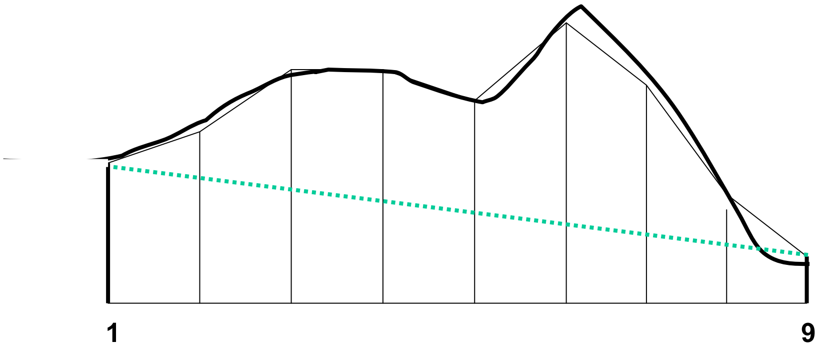
Individual trapezoid approximation:

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

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

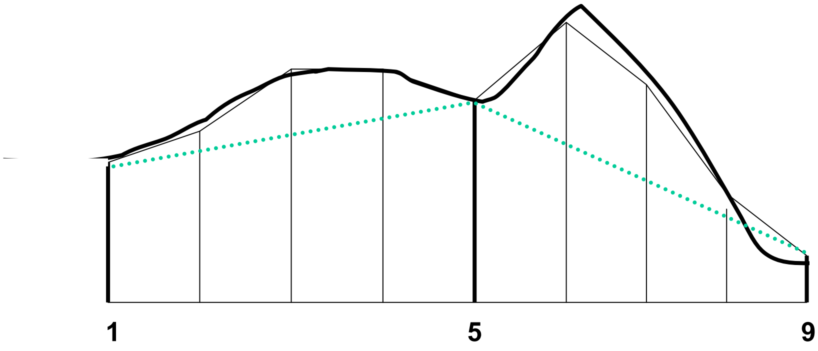
$$\int_{x_1}^{x_N} f(x)dx = h(0.5 f_1 + f_2 + ... + f_{N-1} + 0.5 f_N) + O((b-a)^3 f''/N^2)$$

Better Trapezoid Rule



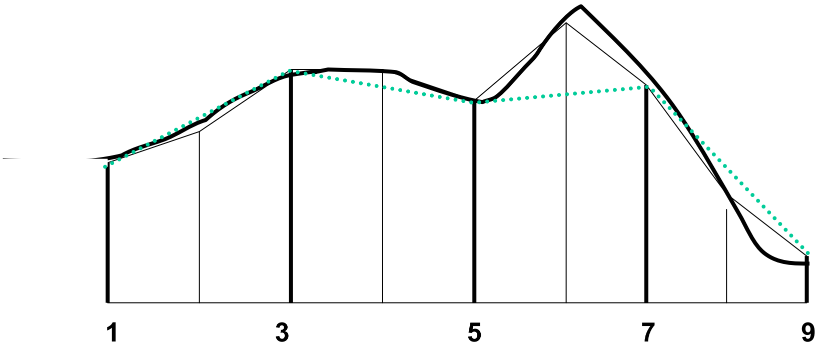
N=1, requires two function evaluations

Better Trapezoid Rule



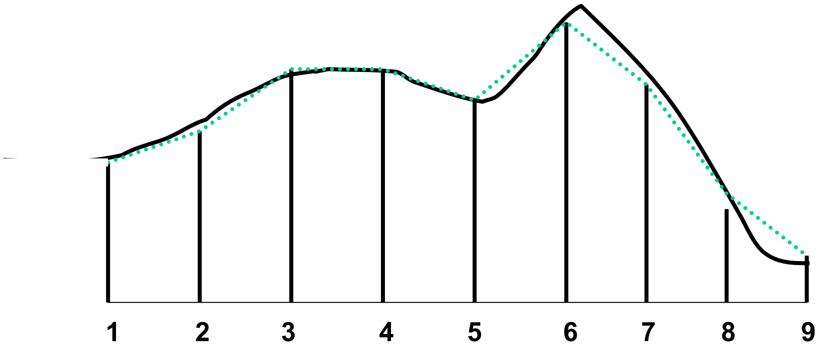
N=2, requires only one more function evaluation

Better Trapezoid Rule



N=4, requires only two more function evaluations

Better Trapezoid Rule



N=8, requires only 4 more function evaluations

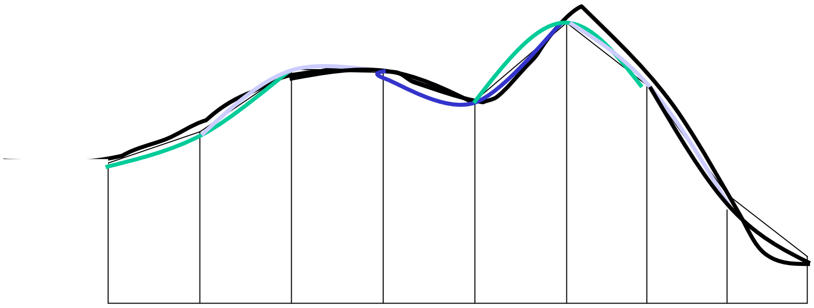
Using Trapezoidal Rule

- 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

```
public 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 delta= (b-a)/tnm; // Spacing of points
            double x= a+0.5*delta;   // 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+= delta; }
            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;         // Value of integral
    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 FuncA(), 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 FuncA(), 0.0, 8.0);
        System.out.println("Integral: " + ans);
        System.exit(0);    }   }            // End Simpson class
```

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

Exercise

- Download Simpson and Trapezoid
  - Run them with different values of m, which governs the size of the interval
    - Explore from m= 5 to m= 20 iterations
    - Number of intervals is 2^{m+1}
    - 2^{20} is about a million
  - Notice that Simpson is much more accurate with many times fewer iterations

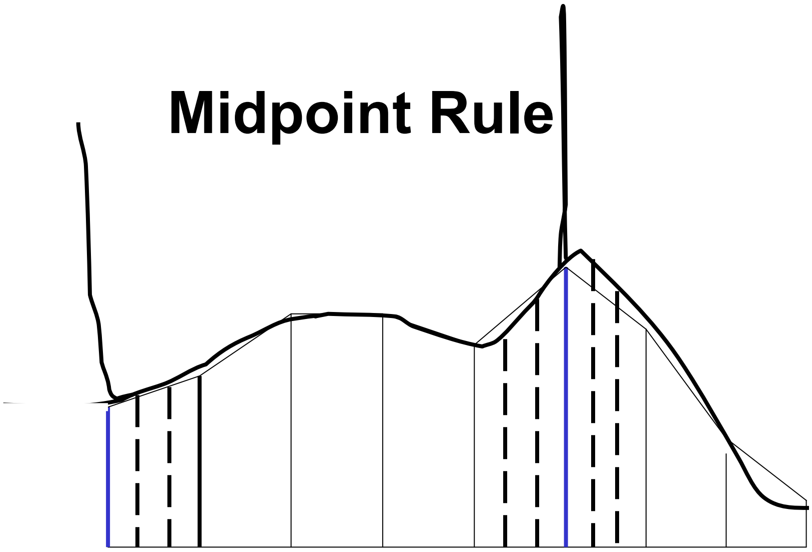
Romberg Integration

- 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