

1.00/1.001 Tutorial 8

October 31 & November 1, 2005

Topics

- 2D API Review
- Affine Transformations
- Numerical Methods:
 - Integration
 - Root Finding
 - Matrices
- Problem Set 7

2D API Review - First Things First

- Invoke `super.paintComponent(g)`
- Cast `g` to a `Graphics2D` object

```
public void paintComponent(Graphics g) {  
  
    super.paintComponent(g);  
    Graphics2D g2 = (Graphics2D)g;  
  
    // Start drawing  
  
}
```

2D API Review - What Can We Draw?

- **String**

```
Font myFont = new Font("Arial", Font.BOLD, 12);  
g2.setFont(myFont);  
g2.drawString("Draw This", 100, 200);
```

- **Line**

```
g2.setStroke(new BasicStroke(1));  
g2.drawLine(200, 300, 400, 500);
```

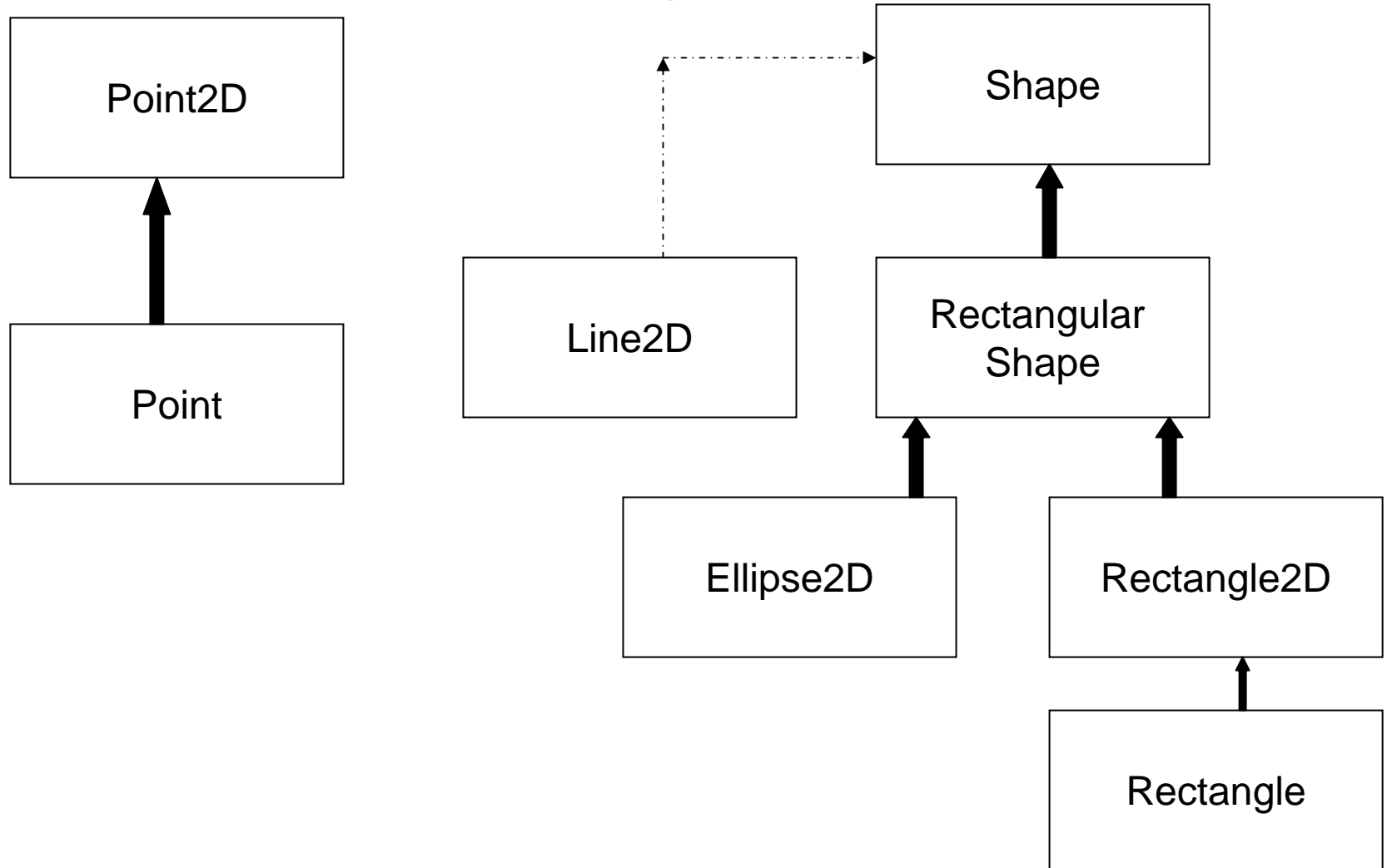
- **Shape (interface)**

- Known implementing classes:

Line2D, Rectangle2D, Ellipse2D

```
Shape s = new Rectangle2D.Double(10, 10, 20, 30);  
Shape c = new Ellipse2D.Double(30, 40, 10, 10);  
g2.draw(s);  
g2.fill(c);
```

2D API Review - What Can We Draw?

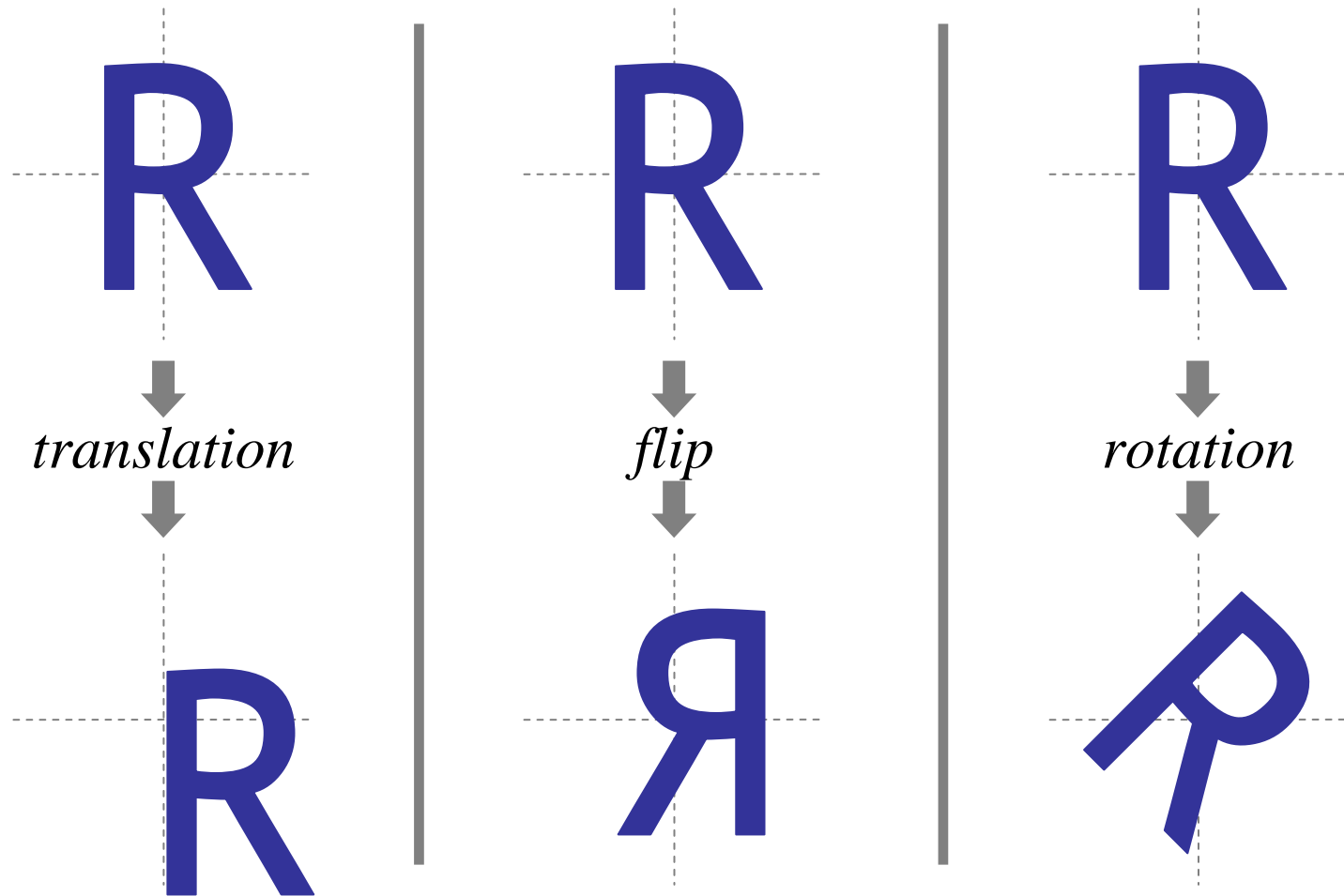


Affine Transformations

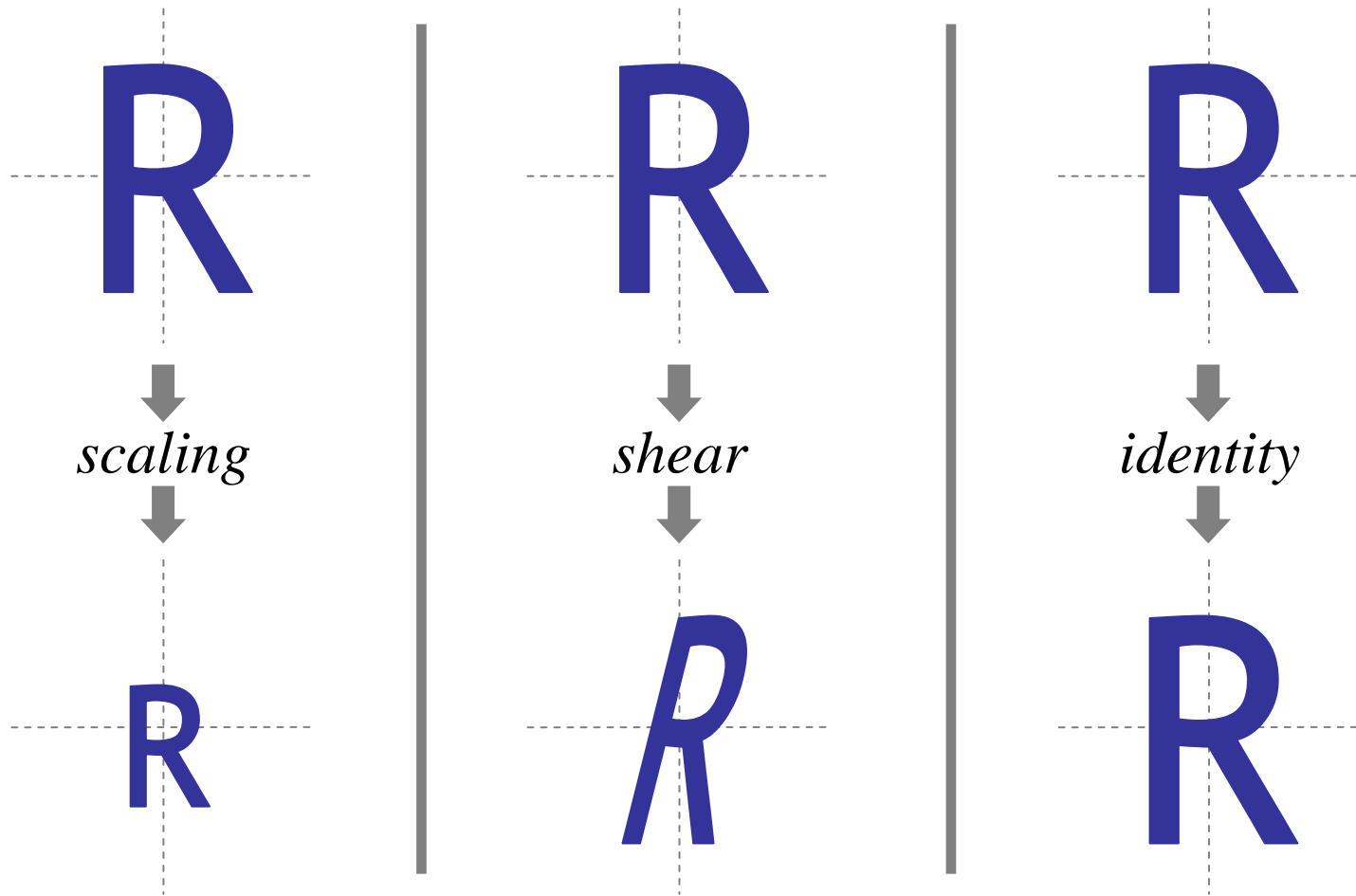
from Sun's Javadoc for `AffineTransform`

- A linear mapping from 2D coordinates to another set of 2D coordinates that preserves the "straightness" and "parallelness" of lines.
- Affine transformations can be constructed using sequences of translations, scales, flips, rotations, and shears.

Affine Transformations



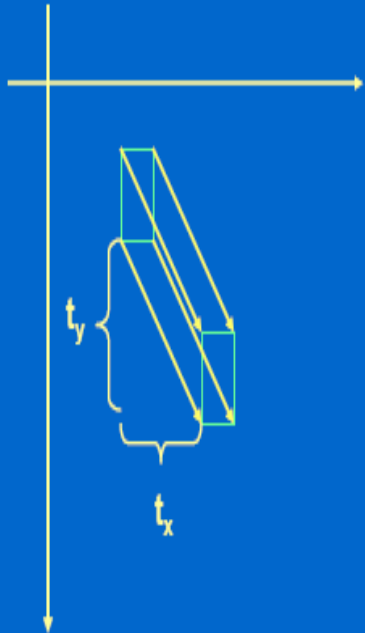
Affine Transformations



Affine Transformations

- An Affine Transform simply encapsulates a 3 x 3 matrix for a given transformation.
- Approaches:
 1. Use AffineTransform's `createTransformedShape` method to create a new, transformed shape from an old one. Then you can draw the shape using Graphics' `draw` and/or `fill` methods.
 2. Can also apply AffineTransform to the Graphics2D object.

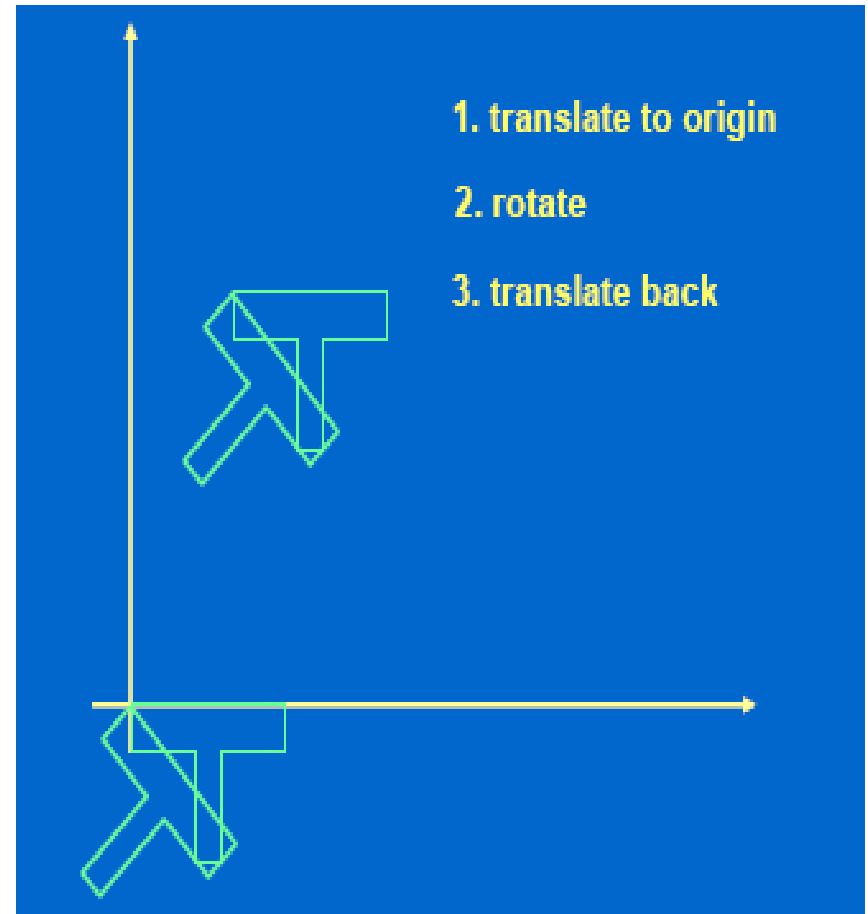
Translation


$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+t_x \\ y+t_y \\ 1 \end{bmatrix}$$

Source files called `TranslatePanel.java`, `ScalePanel.java`, `RotatePanel.java` and `TransformMain.java`

Affine Transformations - Review

- When we transform a shape, we transform each of the defining points of the shape, and then redraw it.
- If we scale or rotate a shape that is not anchored at the origin, it will translate as well.
- If we just want to scale or rotate, then we should translate back to the origin, scale or rotate, and then translate back.



Source files called `TranslatePanel.java`, `ScalePanel.java`, `RotatePanel.java` and `TransformMain.java`

Transformation Exercise

- Modify `scalePanel.java` and `rotate.java` to transform around anchor points

Numerical Methods: Question

$$f(x) = x^2 + x + 1$$

- Which of the following choices can be used to represent this function in Java?
 - a) An array or ArrayList
 - b) Return value of a method
 - c) A class
 - d) An interface

Numerical Integration & Root Finding

- Packaging mathematical functions as objects
 - In some languages methods can be passed around as arguments
 - In C and C++ done using “Function Pointers”
 - Java does not directly allow this, so we have to fake it: wrap the method inside a class or interface
 - So, instead of writing a method called `max` and passing it around by name, write a class that has a `max` method, and pass around objects of that class.
- It's good practice to have classes that represent functions implement a common interface
 - Why? Suppose we have an algorithm that is general to all 1-D functions, we need to implement it *only once* – makes code maintainable and portable

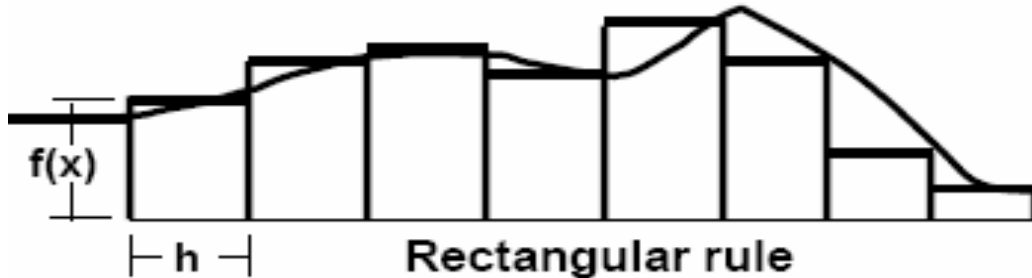
Numerical Integration

- Basic problem: Given $f(x)$, evaluate

$$\mathbf{F} = \int_{\mathbf{x}=\mathbf{a}}^{\mathbf{x}=\mathbf{b}} \mathbf{f}(\mathbf{x}) \mathbf{d}\mathbf{x}$$

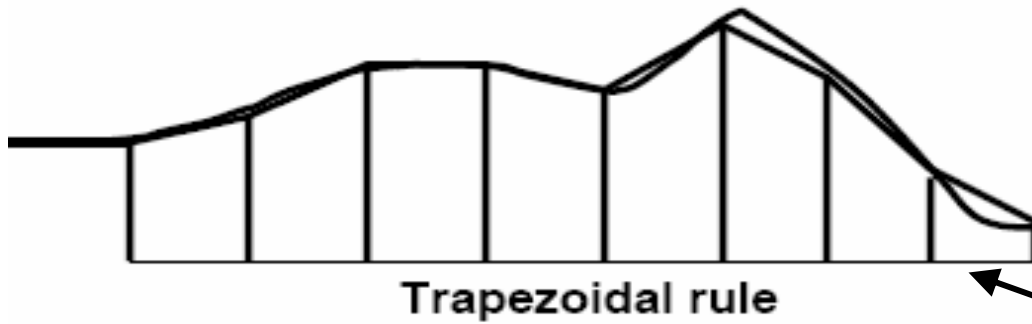
- Approaches:
 - Monte Carlo: Sample $[a,b]$ randomly, add up function values at each sampled point, divide by $b-a$.
 - Woefully inaccurate !! Needs a large number of samples
 - Rectangular/Trapezoidal/Simpson's: Approximate f using piecewise constant, linear, parabolic segments and integrate each segment individually
 - More “elite” algorithms like Gauss quadrature

Numerical Integration



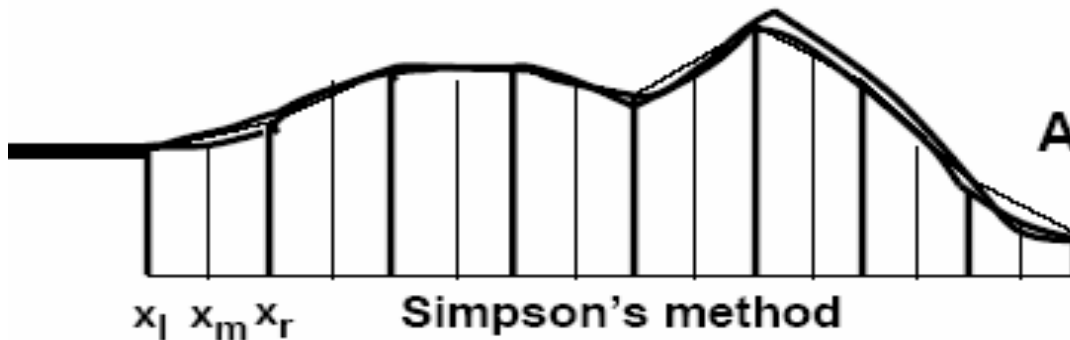
$$A = f(x_{\text{left}}) * h$$

Almost never
used



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

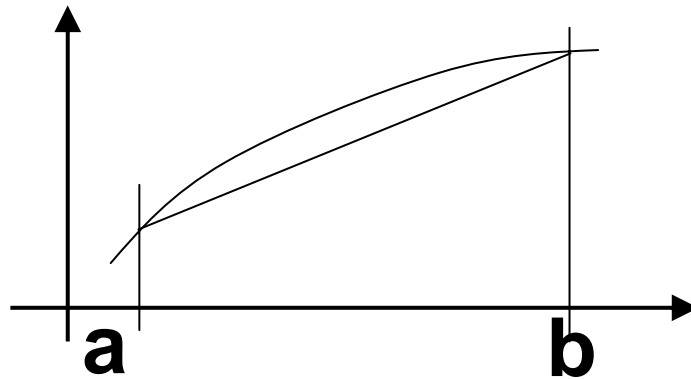
Use fancier version
covered in the
lecture



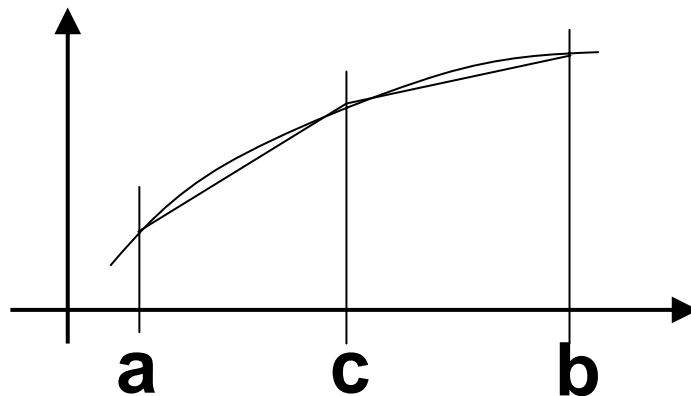
$$A = (f(x_l) + 4f(x_m) + f(x_r)) * h / 6$$

Improved Trapezoidal Rule

- Basic Idea: Divide and Conquer



$$I_0 = \frac{1}{2} (f(a) + f(b)) \cdot h$$



$$I_1 = \frac{1}{2} (f(a) + f(b)) \cdot \frac{h}{2} + f(c) \cdot \frac{h}{2}$$

$$= I_0 \cdot \frac{1}{2} + f(c) \cdot \frac{h}{2}$$

- Hence, to compute integral at one subdivision, need to know integral at previous subdivision – therefore need to call `trapzd` multiple times (Can use recursion instead of a `for` loop)

Integration Exercise

- Compute PI using “fancy” Trapezoidal rule
 - Recall calling convention !
 - Why not using the regular Trapezoidal rule?
 - Hint: More accurate? Fewer operations? Recommended by TA?

$$\pi = \int_{\mathbf{x}=0}^{\mathbf{x}=1} \frac{4}{1 + \mathbf{x}^2} d\mathbf{x}$$

- Implement `MathFunction` in `FuncPI.java`
- Complete `main()` in `ComputePI.java`
- How accurate is your estimate? How does it converge with the number of intervals?

Root Finding Methods

Two major types:

- **Bracketing methods:** solution must be known to lie in a particular interval. Always converge to the value of a root in that case.
 - Bisection, False Position
- **Open methods:** use one or more initial guesses, but it's not necessary to know the interval in which a solution lies. Not guaranteed to converge to a solution.
 - Fixed point iteration, Secant, Newton-Raphson

Bracketing Methods: Bisection

- Reduce the size of the bracket by half after every iteration, until the sides are close enough to suit our preference.
- Decide which of the 2 brackets to move towards midpoint based on whether $f(\text{lower}) * f(\text{midpt})$ is $>$, $<$, or $== 0$.
- If there are 2 roots in the interval, finds only one of them!

Source files called `MathFunction.java`, `FuncA.java` and `RootFinder1.java`

Bracketing Methods: Bisection

```
public static double bisection(MathFunction1D func, double x1,
    double x2, double epsilon)
{
    double m;
    while(Math.abs(x1-x2)>epsilon)
    {
        m=(x1+x2)/2.0;
        if(func.f(x1)*func.f(m)<=0.0)
            {x2=m;} //root in left subinterval, rt. Bracket moves left
        else
            {x1=m;} //root in rt. Subinterval, left bracket moves right
    }
    return m;
}
```

Bracketing Methods: False Position

- Bisection method ignores magnitudes of $f(x_1)$ and $f(x_2)$.
 - If $f(x_1)$ is -0.5 , and $f(x_2)$ is 12.3 , root is probably closer to x_1 . But bisection method guesses halfway between x_1 and x_2 .
- Connect $f(x_1)$ and $f(x_2)$ with a straight line; where it crosses the x-axis is the new estimate x_{est} . Is $f(x_1) \cdot f(x_{est}) >$, $<$, or equal to 0? Again, keep the interval in which the sign change occurs.
- Same algorithm as bisection: just different estimate x_{est} for the root at each step. Usually, but not always better than bisection.

Bracketing Methods: False Pos.

```
public static double falsePos(MathFunction1D func, double x1,
    double x2, double epsilon)
{
    double xEst;
    while(Math.abs(x1-x2)>epsilon)
    {
        xEst=x2-((func.f(x2)*(x1-x2))/(func.f(x1)-func.f(x2)));
        if(func.f(x1)*func.f(xEst)<=0.0)
            {x2=xEst;} //root in l. subinterval, rt. Bracket moves left
        else
            {x1=xEst;} //root in rt. subinterval, lt. bracket moves right
    }
    return xEst;
}
```

Open Methods: Secant

- Only initial guesses needed: no bracketing!
- For 1D functions, new guess x_{new} is the x -crossing of a tangent line from $(x_{\text{old}}, f(x_{\text{old}}))$.
- This requires the derivative of $f(x)$. In secant, however, the derivative is not available.
- Approximates derivative with a finite divided difference.
- Requires 2 initial estimates, but they do not have to bracket solution
- Usually converges quickly, oscillating around solution.

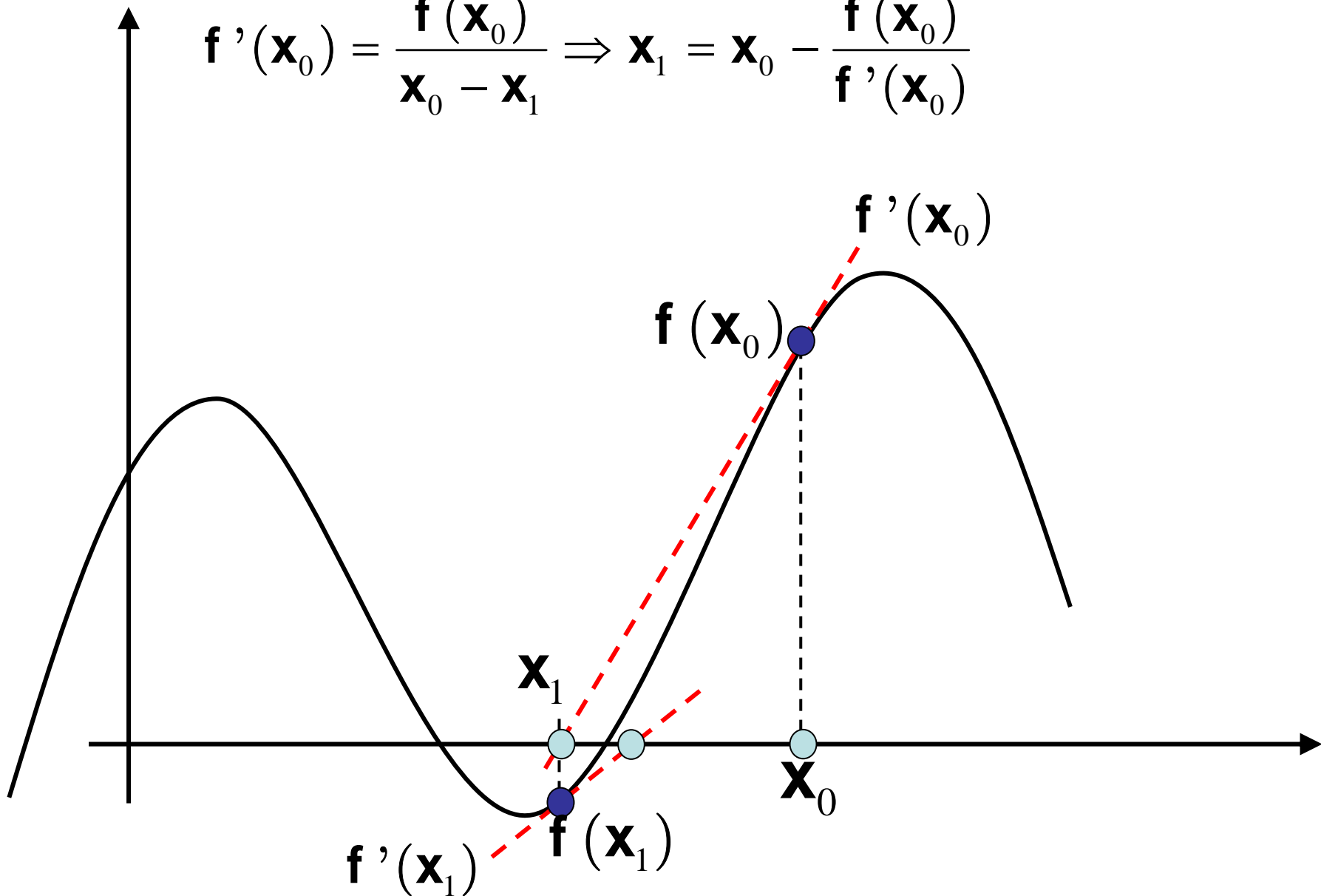
Secant Method: Algorithm

- For the first iteration it assumes that this approximation is the line that connects 2 points (x_1 & x_2) of an interval
- It then calculates x_3 as the point where this line crosses the x-axis
- The next iteration draws a line between x_3 & x_2 . x_4 is then calculated as the point where this line crosses the x-axis
- Requires 2 initial estimates, but they do not have to bracket solution
- Usually converges quickly, oscillating around solution.

Newton-Raphson

- Only one initial guess needed: no bracketing
- For 1D functions, new guess x_{new} is the 0-crossing of a tangent line from $(x_{\text{old}}, f(x_{\text{old}}))$. This requires the derivative of $f(x)$. For >1D functions all first order partial derivatives required.
- Usually converges quickly, oscillating around solution, provided initial guess is good
- Requires change of interface for functions to be solved by Newton's method... what change, and why?

$$f'(x_0) = \frac{f(x_0) - f(x_1)}{x_0 - x_1} \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

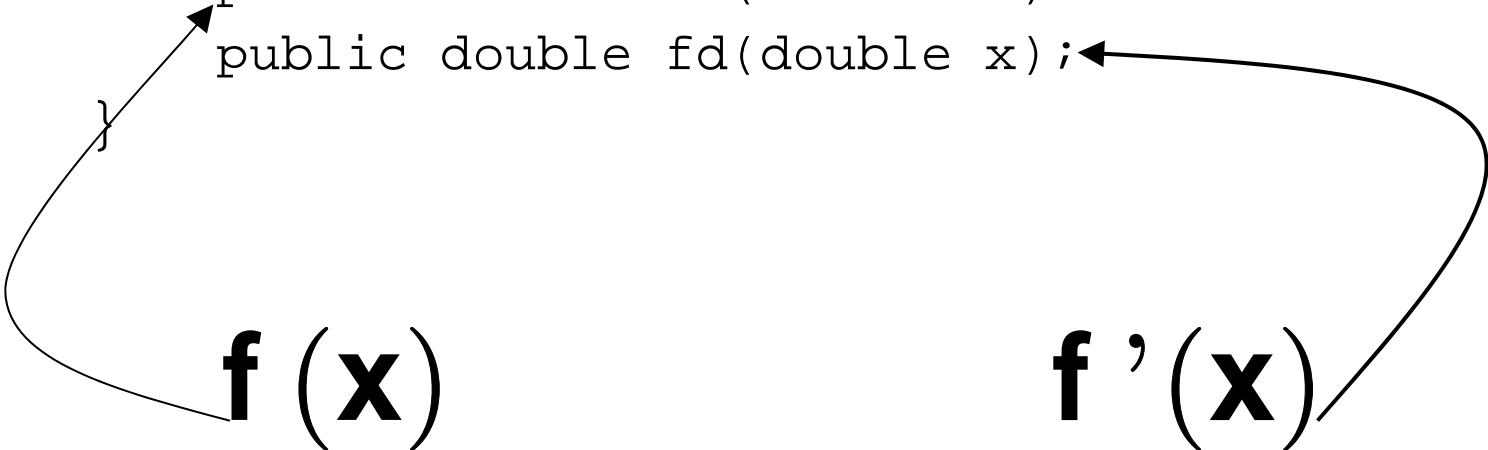


Interface for functions solved using Newton-Raphson

```
public interface MathFunction2 {  
    public double fn(double x);  
    public double fd(double x);  
}
```

f (x)

f ' (x)



Exercise 2: Integration

- Compute PI using “fancy” Trapezoidal rule
 - Recall calling convention !
 - Why not using the regular Trapezoidal rule?
 - Hint: More accurate? Fewer operations? Recommended by TA?

$$\pi = \int_{\mathbf{x}=0}^{\mathbf{x}=1} \frac{4}{1 + \mathbf{x}^2} d\mathbf{x}$$

- Implement `MathFunction` in `FuncPI.java`
- Complete `main()` in `ComputePI.java`
- How accurate is your estimate? How does it converge with the number of intervals?

Root Finding Exercise

- When is $\mathbf{x} = \cos(\mathbf{x})$?
- Implement `MathFunction2` in `NewtonFunc.java`
- Complete the `main()` method in `NewtonTest.java`
- How many Newton iterations does it take? What is the final error in the root?
- Optional Exercise: Compare with Bisection/Fixed point iteration

Matrices & Linear Systems

- Matrices often used to represent a set of linear equations

$$\begin{aligned}
 a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \dots + a_{0,n-1}x_{n-1} &= b_0 \\
 a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \dots + a_{1,n-1}x_{n-1} &= b_1 \\
 \dots & \\
 a_{m-1,0}x_0 + a_{m-1,1}x_1 + a_{m-1,2}x_2 + \dots + a_{m-1,n-1}x_{n-1} &= b_{m-1}
 \end{aligned}$$

- Coefficients a and right hand side b are known

$$\begin{array}{ccccc|c|c}
 a_{00} & a_{01} & a_{02} & a_{03} \dots & a_{0,n-1} & x_0 & b_0 \\
 a_{10} & a_{11} & a_{12} & a_{13} \dots & a_{1,n-1} & x_1 & b_1 \\
 a_{20} & a_{21} & a_{22} & a_{23} \dots & a_{2,n-1} & x_2 & b_2 \\
 \dots & \dots & \dots & \dots \dots & \dots & \dots & \dots \\
 a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & a_{m-1,3} \dots & a_{m-1,n-1} & x_{n-1} & b_{m-1}
 \end{array} =$$

(m rows x n cols)

(n x 1) = (m x 1)

$$\mathbf{Ax} = \mathbf{b}$$

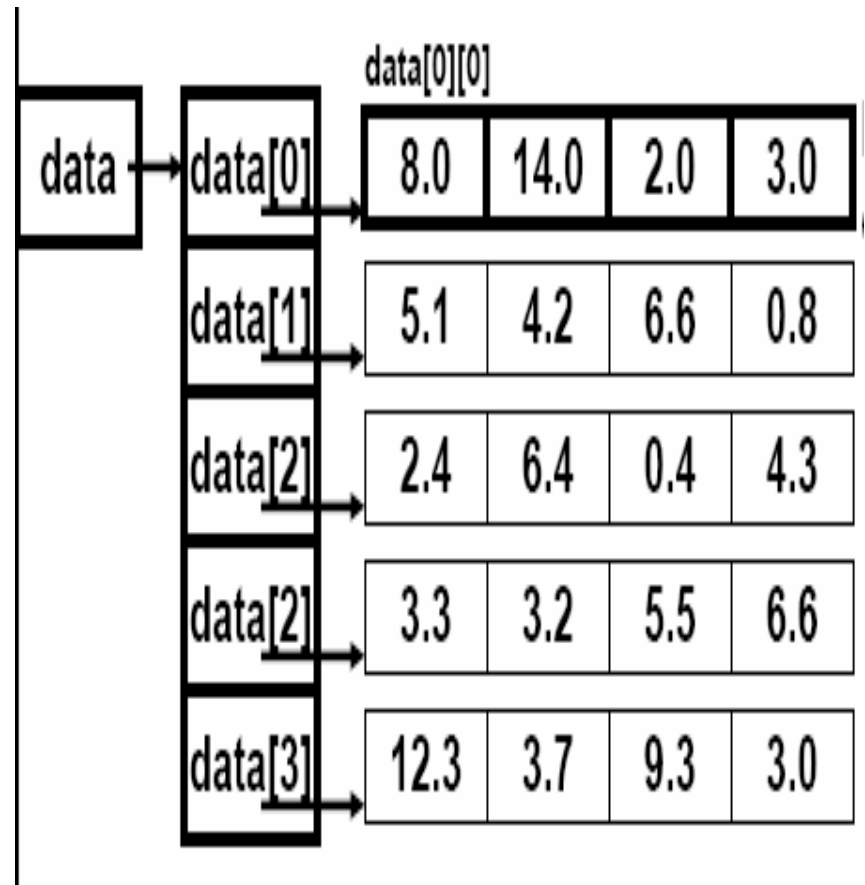
- n unknowns x related to each other by m equations

Matrices & Linear Systems

- If $n=m$, we will try to solve for unique set of x .
- Obstacles:
 - If any row (equation) or column (variables) is a linear combination of others, matrix is degenerate or not of full rank. No solution.
 - If rows or columns are nearly linear combinations, roundoff errors can make them linearly dependent. Failure to solve although solution might exist.
 - Roundoff errors can accumulate rapidly. While you may get a solution, when you substitute it into your equation system, you'll find it's not a solution.
- JAVA has 2D arrays for defining matrices. However, are no built-in methods for them

Matrices & Linear Systems

- Create Matrix classes and have methods for adding, subtracting, multiplying, forming identity matrix etc.
- A 2-D array is a reference to a 1-D array of references to 1-D arrays of data. This is how matrix data is stored in class Matrix.



Source Files called `Matrix.java` and `MatrixMain.java`

Diagonal Matrices

- Only diagonal elements are non-zero.
- Store this as a matrix of 1-D elements.
- Implement the same methods as Matrix class.
- Does not extend Matrix class, replaces data representation.

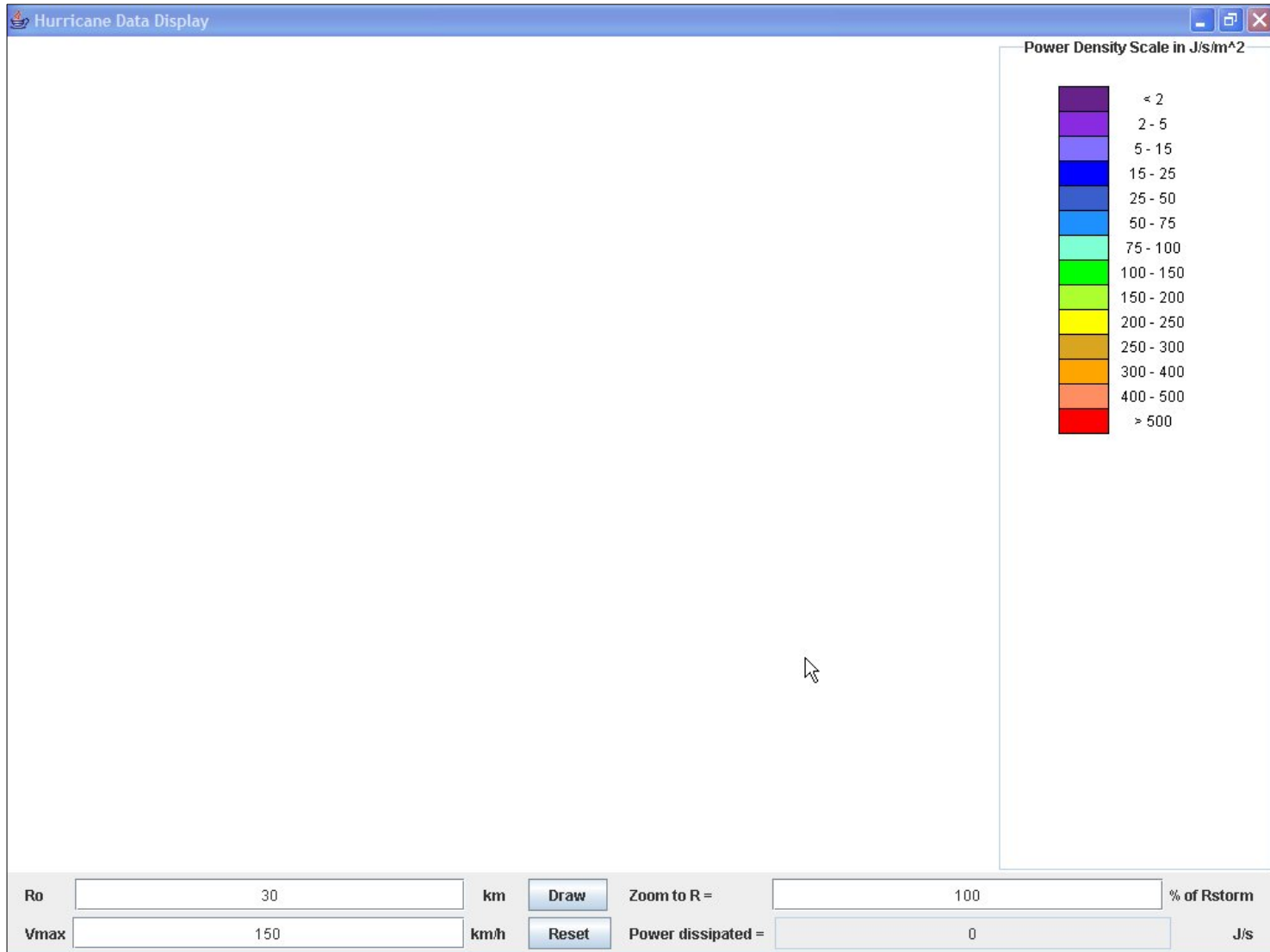
$$\begin{array}{cccccc} | & a_{00} & 0 & 0 & 0 & \dots & 0 \\ & 0 & a_{11} & 0 & 0 & \dots & 0 \\ & 0 & 0 & a_{22} & 0 & \dots & 0 \\ & \dots & \dots & \dots & \dots & \dots & \dots \\ & 0 & 0 & 0 & 0 & \dots & a_{m-1,m-1} \\ | & & & & & & \end{array}$$

Source file called [DiagonalMatrix.java](#)

Problem Set 7

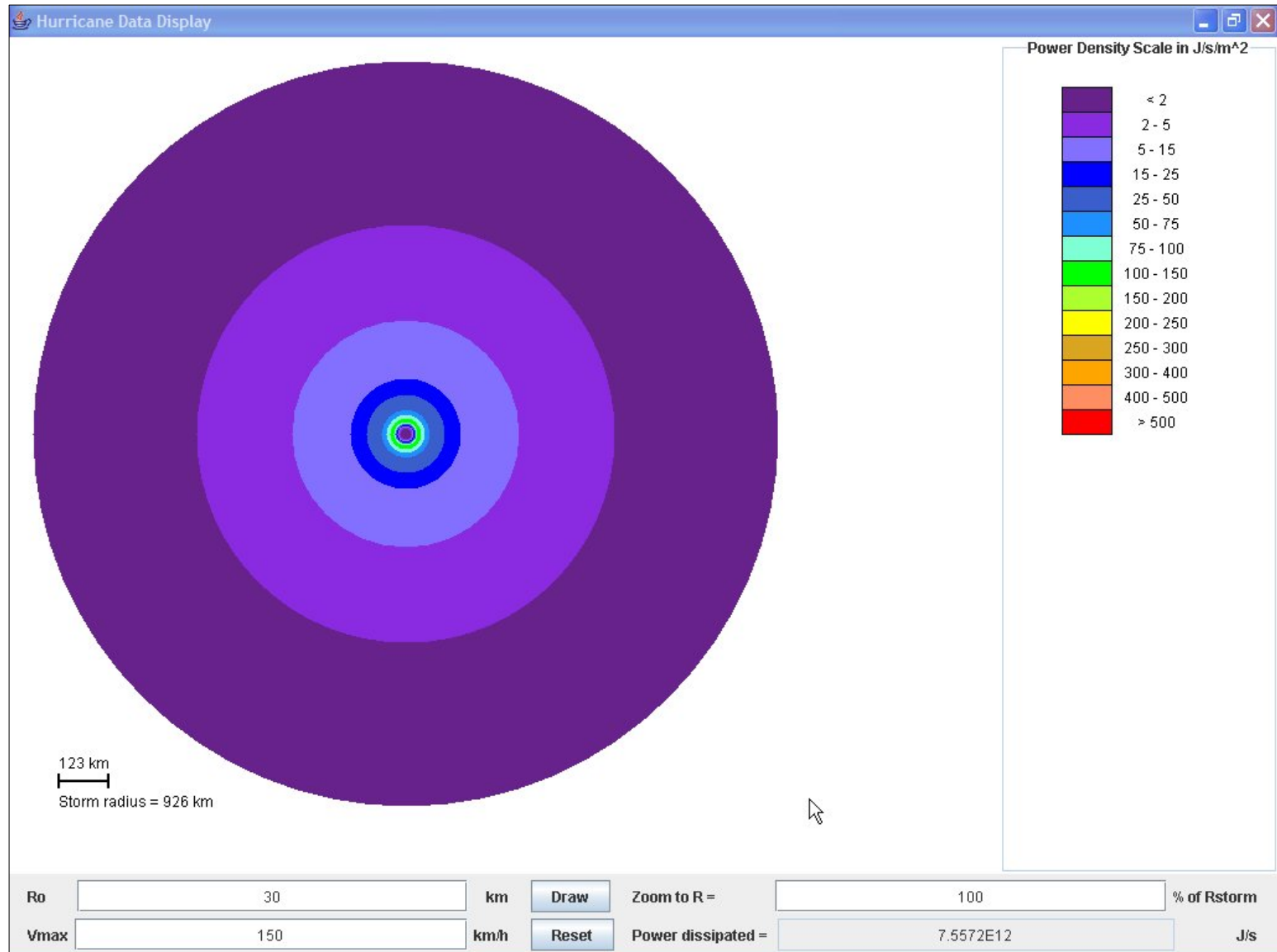
- Find the power dissipated by a hurricane and model the hurricane power density visually with a GUI.
- Make sure you check for correct input values
 - Is input empty?
 - Is the input numeric?
 - Is it within the correct range of values?
- We provide you with the equations in the problem set. You should have a method for each equation.
- Think of how many classes you need for the entire problem. Don't put everything into 1 class!
- Step 1: calculations
- Step 2: draw the power density of the storm as concentric circles

PS7 – suggested GUI: Default or “Reset” button look

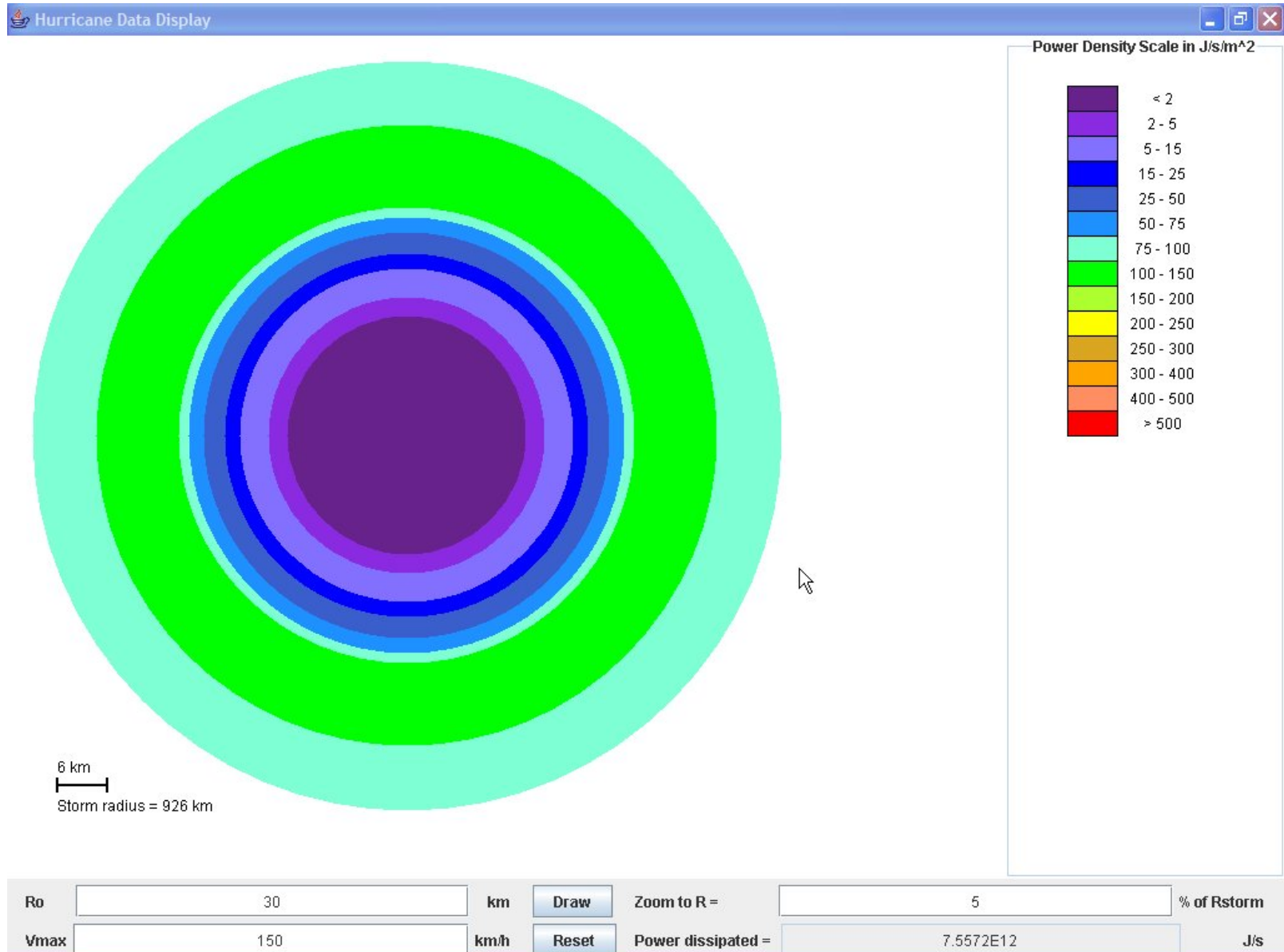


PS7 – suggested GUI

“Draw” button or Full Zoom look



PS7 – suggested GUI: Zoom 5%



PS7 – suggested GUI: Zoom 50%

