1.00 Tutorial 9

Matrices, Integration, Root Finding

Matrices & Linear Systems

- Matrices often used to represent a set of linear equations
- Matrix **A** and right-hand side *b* are known
- *n* unknowns *x* related to each other by *m* equations

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

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

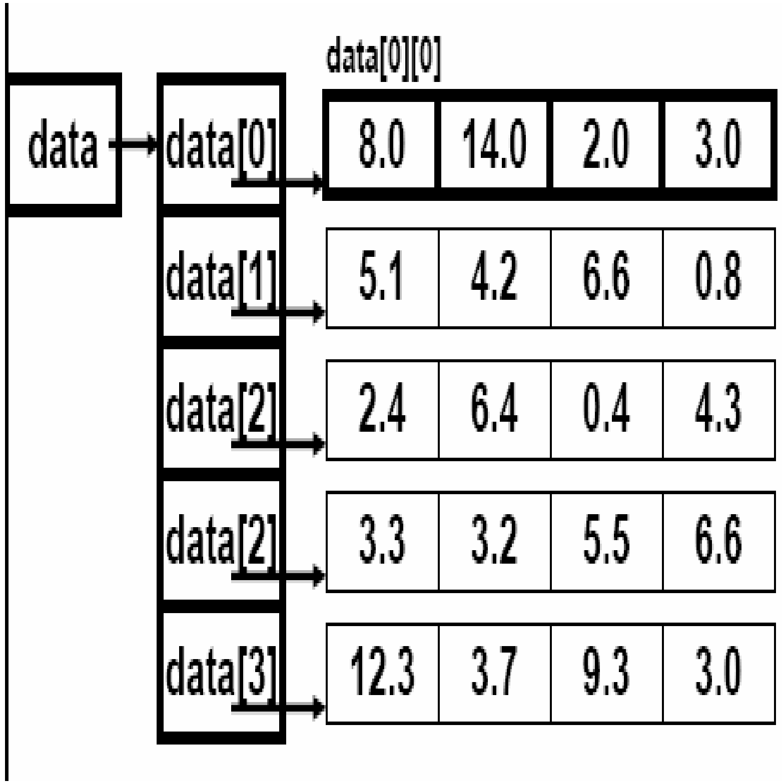
(m rows x n cols) (n x 1) = (m x 1)

Ax=b

Matrix Representation

- 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`.



Traversing 2D Arrays

- Say `double[][] data = new double[M][N]`
- `data.length` = number of rows (`M`)
- `data[0].length` = number of columns (`N`)
- Typical operation accessing all elements of a `Matrix`

```
for(int i = 0; i < data.length; i++) {
  for(int j = 0; j < data[0].length; j++) {
      // Do something here
  }
}
```

Matrix exercise

- Add an instance method in the `Matrix` class to extract a `MxN` submatrix of a given matrix starting at position `(I,J)`
- If the input is invalid (section too large, etc.), return `null`
- Use **`Matrix.java`**, **`MatrixTest.java`**

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

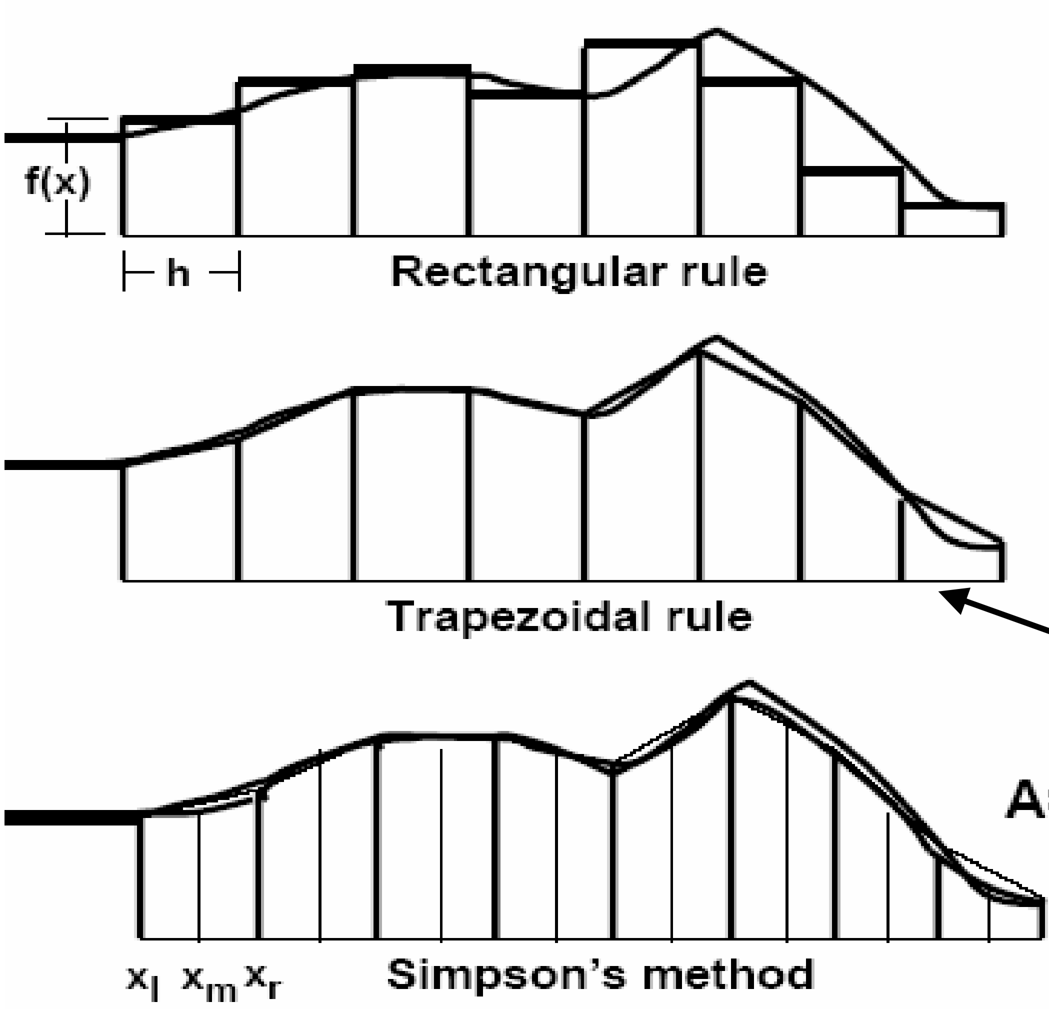
Numerical Integration

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

$$F = \int_{x=a}^{x=b} f(x)\, dx$$

- 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



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

Almost never used

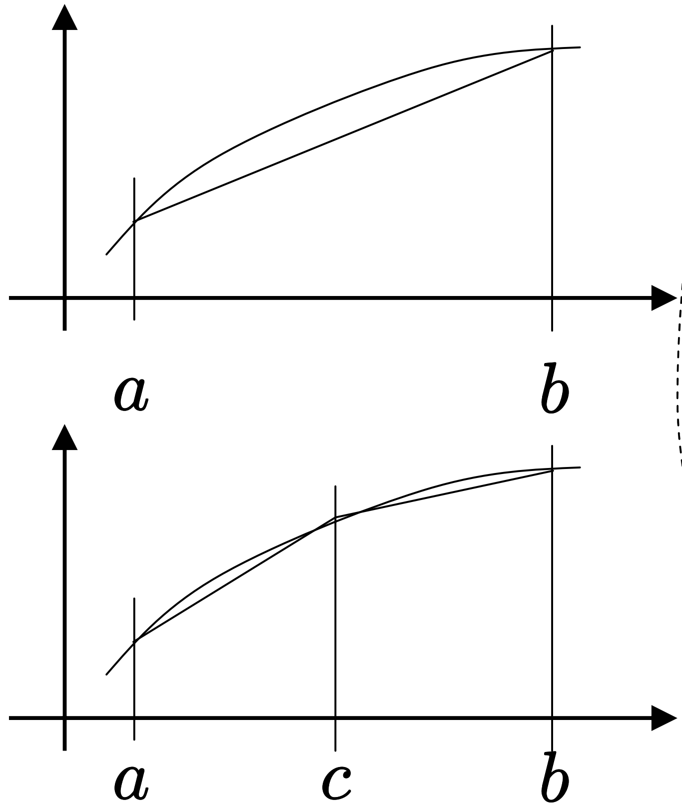
$A = (f(x_{left}) + f(x_{right})) * h/2$

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



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

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

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

- 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?

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

- 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

Newton-Raphson

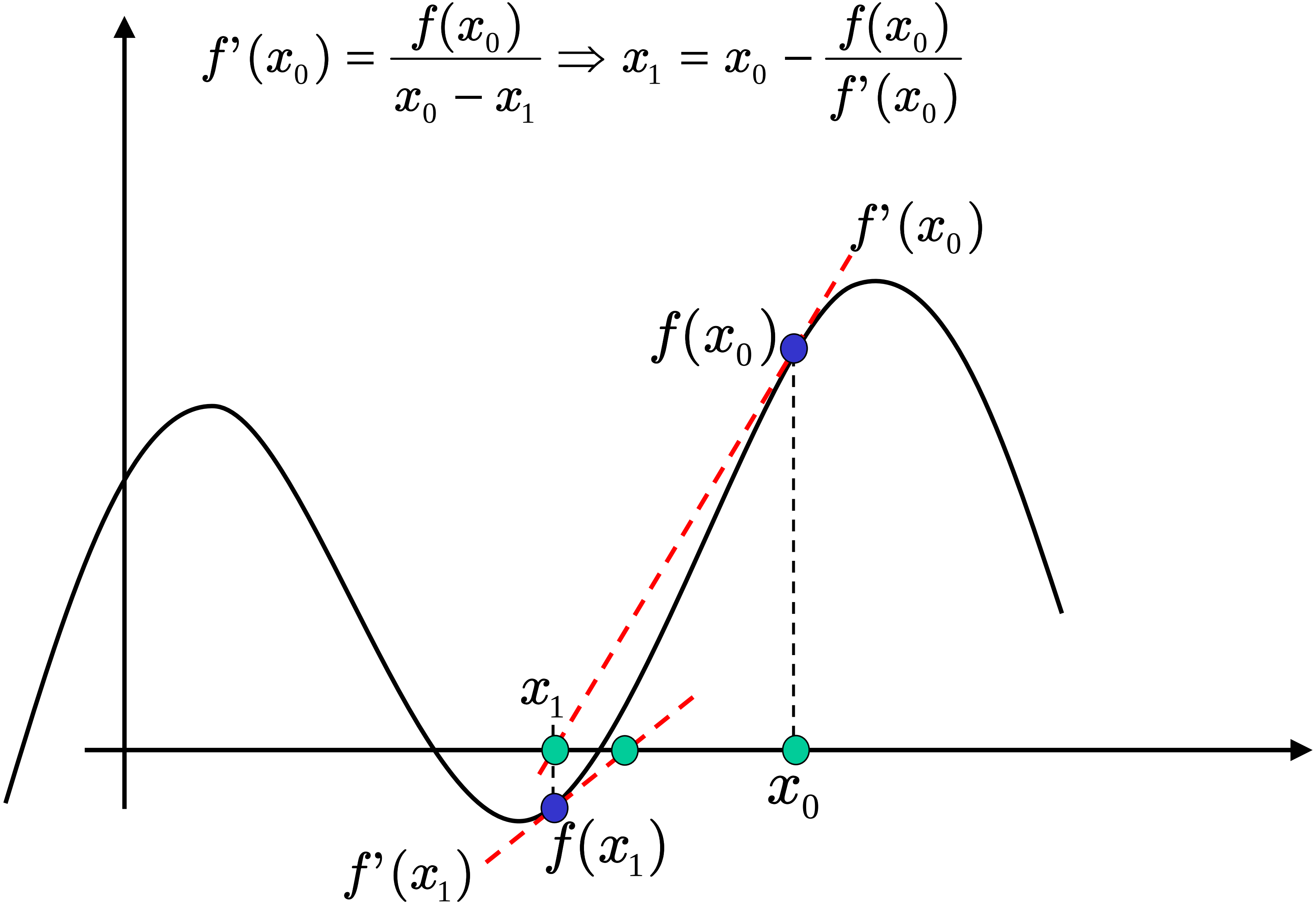
•& Only one initial guess needed: no bracketing

•& For 1D functions, new guess x_{new} is the 0-crossing of a tangent line from (x_{old}, $f(x_{\mathrm{old}})$). 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?

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



Interface for functions solved using Newton-Raphson

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

$f(x)$ $f'(x)$

Root Finding Exercise

- When is $x = \cos(x)$?
- 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

PS 8: Numerical Methods

- Numerical integration (Sailboat Mast)
  - Given

    $$f(z) = 200\left(\frac{z}{z+5}\right)e^{-2z/30}$$

  - Need to compute

    $$F = \int_0^{30} f(z)dz \qquad d = \frac{\int_0^{30} z f(z)dz}{\int_0^{30} f(z)dz}$$

  - Use Rectangular, Trapezoidal and Simpson's rules
  - Treat Simpson's rule as accurate and see how much the other methods differ from it.

Approach

- Write two classes implementing `MathFunction`, one for $f(z)$ and other for $z*f(z)$
- In the main method, for each of the three integration schemes, evaluate $f(z)$ and $z*f(z)$ by passing the classes you implemented to the respective integration methods
- For Rectangle rule , use `Integration.Rect` in file `Integration.java`
- For Trapezoidal rule, use `Trapezoid.trapzd` in file `Trapezoidal.java` [remember calling convention!]
- For Simpson's rule, use `Simpson.qsimp` in file `Simpson.java`

PS 8: Numerical Methods

- Root finding (Open channel flow)
  - Find a d such that

$$f(d) = \frac{\sqrt{s}}{n}\left(\frac{(dw)^{5/3}}{(d+2w)^{2/3}}\right) - Q = 0$$

  - Use bisection and Newton iterations
    - Need to modify `RootFinder.rtbis` and `Newton.newt` to output the number of iterations
  - Key problems
    - Implementing $f(d)$
    - Validating the input (Cannot have negative values!)
    - Reasonable choices for initial guesses ??

Approach

- Implement `MathFunction` (for bisection) and `MathFunction2` (for Newton)
  - Hint: Can write *one* class implementing *both* interfaces (cf. multiple inheritance)
  - Expressions given in the pset
- In the `main()` method:
  - Get inputs from using `JOptionPane`
  - Simply pass in objects of your class into the solvers and solve for d
  - Compute velocity, flow rate, etc.

PS 8: Numerical Methods

- Optional GUI