Linear Equations

Previously learned how to solve an equation with one unknown. For an equation like:

$$10 = 5 \cdot x + 2$$

- A series of operations are performed on each side of the equation.
- By performing identical operations on the left and right side of the equation, equality is maintained.

$$\begin{aligned} 10 &= 5 \cdot x + 2 \\ 10 - 2 &= 5 \cdot x + 2 - 2 \\ \frac{1}{5}\{10 - 2\} &= (5 \cdot x + 2 - 2)\,\frac{1}{5} \\ \frac{8}{5} &= x \end{aligned}$$

- similar process used to recast an equation in terms of a different variable
- example: using Darcy's Lay to determine the hydraulic conductivity (K)

$$Q = -K \cdot A \cdot \frac{dh}{dl}$$

$$K = -\frac{Q}{A\frac{dh}{dl}}$$

Simultaneous Equations

A series of equations can be solved if:

- their are an equal number of equations and unknowns
- the equations are linearly independent

$$\begin{aligned} 2x + 4y &= 11 \\ 3x + 7y &= 15 \end{aligned}$$

To solve for x and y, on equation is recast in terms of an unknown variable, then substituted into the remaining equation.

$$\begin{aligned} 2x &= 11 - 4y \\ x &= \frac{11}{2} - 2y \\ 3(\frac{11}{2} - 2y) + 7y &= 15 \\ \frac{33}{2} - 6y + 7y &= 15 \\ y &= \frac{-3}{2} \\ 2x &= 11 - 4(\frac{-3}{2}) \\ 2x &= 17 \\ x &= \frac{17}{2} \end{aligned}$$

- Going through these manipulations for many equations is tedious and error-prone.
- An alternative way of doing this is to rewrite the equations in matrix notation.

Again, consider the equations:

$$\begin{aligned} 2x + 4y &= 11 \\ 3x + 7y &= 15 \end{aligned}$$

- Solving these equations by writing in matrix form and using Gaussian Elimination.
- essentially an accounting system for the algebra

$$\begin{bmatrix} 2 & 4 \\ 3 & 7 \end{bmatrix} \begin{bmatrix} 11 \\ 15 \end{bmatrix}$$

The goal of Gaussian Elimination is to:

- manipulate the matrix until it has the form of an *upper triangular* matrix.
- done by multiplying lines by a constant and adding or subtracting one line from another.

Multiplying row 1 by 1.5:

$$\begin{bmatrix} 3 & 6 \\ 3 & 7 \end{bmatrix} \begin{bmatrix} 16.5 \\ 15 \end{bmatrix}$$

Subtracting row 1 from row 2:

$$\begin{bmatrix} 3 & 6 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 16.5 \\ -1.5 \end{bmatrix}$$

Implying that $y = -1.5$. Back-substitution can again be used to find x.

- The steps for Gaussian Elimination can be formulated into a matrix format.
- The steps were: [1] multiply row 1 by 1.5 and [2] subtract row 1 from row 2.

Putting this in matrix format:

$$\begin{bmatrix} 1 & 0 \\ -1.5 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix}$$

- The matrix A can now be factored into a lower and upper triangular parts.
- In previous example a lower triangular matrix was multiplied by 'A' to form an upper triangular matrix.
- The inverse of this lower triangular matrix is 'L'

$$LU = A$$

Writing out the matrix equation in pieces:

$$\begin{aligned} Ux &= c \\ Lc &= b \\ LUx &= b \end{aligned}$$

Because the matrices are triangular, finding **c** and **x** by forward/back substitution is easy. Once a matrix is factored, answers for **x** for different values of **b** are simple to calculate.

- method is a workhouse for solving simultaneous linear equations
- number of operations for Gaussian Elimination are around $\frac{n^3}{3}$.
- Once factored into **L** and **U**, takes about $\frac{n^2}{2}$ operations
- 10 equations with 10 unknowns will be 10 time faster using LU decomposition, once the factoring is done.

Gaussian Elimination

```
import numpy as np

A=[[np.random.random_integers(10) for i in range(3)] for j in range(3)]
BO=np.array(A)+10.*np.identity(3)
B=np.array(A)+10.*np.identity(3)

mtrx_ops=[]
L=np.identity(3)
for i in range(3-1):#cols
    for j in range(i+1,3):#row below diag.
        Z=np.identity(3)
        Z[j,i]=-B[j,i]/B[i,i]
        B=np.dot(Z,B)
        L=np.dot(Z,L)
        mtrx_ops.append(Z)

##to get orignal matrix back

X=np.dot(mtrx_ops[2],mtrx_ops[1])
X=np.dot(X,mtrx_ops[0])

B1=np.dot(X,BO)

'''
to get LU Decomp from scipy
from scipy import linalg
LU=linalg.lu(BO)
'''
```

Matrix Properties

- matrix multiplication $A \cdot B$: rows times columns, number of columns in A must equal rows in B
- matrix multiplication not commutative: $A \cdot B \neq B \cdot A$
- identity matrix (main diagonal)
- no matrix division, matrix inverse
- matrix operations for multiplying row by scalar, adding one row to another, etc.

Ill-Conditioned Matrix

- system of equations close to not being linearly independent
- estimated based on matrix determinant
- if determinant small (compared with largest elements in matrix), is ill conditions

$$|A| < \|A\| \tag{1}$$

- 'condition number' based on matrix norm

$$\|A\| = \sqrt{\Sigma_i \Sigma_j A_{ij}^2} \tag{2}$$

  - good when close to one
  - singular matrix when infinity (or very large)
- small changes in coefficients cause large changes in solutions

Cramer's Rule

- Determinants(**D**) have many special properties:
  - constant (c) times column in matrix, increases matrix determinant by factor of c
  - adding column times constant to another column doesn't change the determinant

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

multiplying column by constant = x:

$$x \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = \begin{vmatrix} x \cdot a_1 & b_1 \\ x \cdot a_2 & b_2 \end{vmatrix}$$

adding 2nd column times y to first column:

$$x \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = \begin{vmatrix} x \cdot a_1 + y \cdot b_1 & b_1 \\ x \cdot a_2 + y \cdot b_2 & b_2 \end{vmatrix}$$

Cramer's Rule

Method for solving matrix equation.

$$x\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = \begin{vmatrix} x \cdot a_1 + y \cdot b_1 & b_1 \\ x \cdot a_2 + y \cdot b_2 & b_2 \end{vmatrix}$$

rearranging:

$$x\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = \begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}$$

$$x = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

Can find solution vector by substitution solution vector into different columns into matrix, and dividing determinant of this matrix by original determinant.

Determinants

- based on product of permutations

$$\sum_{n!}(-1)^k a_{1_{i_1}} \cdot a_{1_{i_2}} \cdot a_{1_{i_3}} \cdot \dots \tag{3}$$

- product of elements selected from one and only one row and column
- elements ordered according to row number
- k is number of (single) shifts to get col. numbers in order (the number of inversions or pivots)
- alternate way of doing this uses co-factors
  - element multiplied by matrix minor determinant and $(-1)^{row+col}$
  - minor is matrix formed by removing row and column that intersects element.
  - sum these products for each element in one row or column

Determinants

$$\begin{vmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{vmatrix} = 11 \cdot 22 \cdot 33 \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} +$$

$$11 \cdot 23 \cdot 32 \begin{vmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix} + 13 \cdot 21 \cdot 32 \begin{vmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} + 12 \cdot 21 \cdot 33 \begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{vmatrix} +$$

$$13 \cdot 22 \cdot 31 \begin{vmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{vmatrix} + 12 \cdot 23 \cdot 31 \begin{vmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}$$

Determinants

$$\begin{vmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{vmatrix} = (-1)^{1+1} \cdot 11 \cdot \begin{vmatrix} 22 & 23 \\ 32 & 33 \end{vmatrix} +$$

$$(-1)^{1+2} \cdot 12 \cdot \begin{vmatrix} 21 & 23 \\ 31 & 33 \end{vmatrix} + (-1)^{1+3} \cdot 13 \cdot \begin{vmatrix} 21 & 22 \\ 31 & 32 \end{vmatrix}$$

Determinants

- these methods are not very efficient, require lots of calculations
- more efficient method based on three rules associated with determinants
  - a row reduction operation (like done in gaussian elimination) does not change the determinant of a matrix
  - factoring out a value from a row (or column) changes the determinant of the matrix by that factor
  - a triangular matrix has a determinant equal to product of elements along its main diagonal.
- can calculate a determinant by doing operations to make matrix triangular and calculate the product of elements along main diagonal
- if main diagonal reduced to ones, can use factors used convert main daigonal elements to one can be used (product of inverse of these factors)

Tridiagonal systems

- What is a banded matrix?
- Occurs in 1-D models and many other situations
- Thomas Algorithm, based on Gaussian Elimination

letting a_i, b_i, and c_i represent coefficients in 'bands' of matrix, d_i is solution vector, and **first row is unchanged** (i.e. no opertions performed on first row). a is lower and c is upper diagonal. subscripts are row numbers.

$$b_i^{'} = b_i - c_{i-1} \cdot \frac{a_i}{b_{i-1}^{'}}$$
$$d_i^{'} = d_i - d_{i-1}^{'} \cdot \frac{a_i}{b_{i-1}^{'}}$$

Then back substitute: $x_n = \frac{d_n^{'}}{b_n^{'}}$, $x_i = \frac{d_i^{'} - c_i \cdot x_{i+1}}{b_i^{'}}$

Example Script:Thomas Algorithm

```
import numpy as np
num=4

indx=np.arange(num**2)
indx=indx.reshape([num,num])
upper=np.where(indx%(num+1)==1,1,0)
lower=np.where((indx)%(num+1)==4,1,0)

A=np.identity(num,dtype=np.float64)*20. + upper*-3. +lower*-3.
d=np.arange(num,dtype=np.float64)

idx=np.where(upper==1)
c=A[idx]
c=np.hstack((c,[0]))
idx=np.where(np.identity(num)==1)
b=A[idx]
idx=np.where(lower==1)
a=A[idx]
a=np.hstack(([0],a))
x=np.zeros([4])

for i in range(1,num):
    b[i]=b[i]-c[i-1]*a[i]/b[i-1]
    d[i]=d[i]-d[i-1]*a[i]/b[i-1]

x[num-1]=d[num-1]/b[num-2]
for i in range(num-2,-1,-1):
    x[i]=(d[i]-c[i]*x[i+1])/b[i]
```

Iterative methods represent a second group of methods for solving simultaneous linear equations. Point iterative methods split a matrix equation into two pieces:

$$Ax = b$$
$$(M + N)x = b$$
$$Mx = b - Nx$$
$$x = M^{-1}(b - Nx)$$

- A matrix $\mathbf{M}$ is chosen that is easy to invert.
- When $\mathbf{M}$ is the diagonal matrix, the iterative method is called the Jacobi method.
- The vector $\mathbf{x}$ is calculated iteratively,
  - first by making a guess for the values in $\mathbf{x}$
  - then updating the guess using the equation above.
- performance improved dramatically by updating $\mathbf{x}$ as new values for are calculated (Guass-Seidel).
- further performance improvements can be made by 'over-relaxing' the change between each iteration.

Example Script: Point Iterative Methods

```
import numpy as N
import time


A=N.zeros([4,4],'f')
b=N.zeros([4],'f')
x=N.zeros([4],'f')
oldx=N.zeros([4],'f')


for i in range(4):
    for j in range(4):
        if i!=j:
            A[i,j]=i*2+j*3+1
        else:
            A[i,j]=i*4+j*3+1
    b[i]=pow(i,2)/(j+1.)


start=time.time()
error=1.
while error>1.e-9:
    oldx[:]=x[:]
    for i in range(4):
        sumOffDiag=0
        for j in range(4):
            if i!=j:
                sumOffDiag=A[i,j]*oldx[j]
        x[i]=(b[i]-sumOffDiag)/A[i,i]
    error=sum([(x[i]-oldx[i])**2 for i in range(4)])
timechng=time.time()-start
print 'jacobi ', timechng

#raw_input('enter')
x=N.zeros([4],'f')
error=1.
start=time.time()
while error>1.e-9:
    oldx[:]=x[:]
    for i in range(4):
        sumOffDiag=0
        for j in range(4):
            if i!=j:
                sumOffDiag=A[i,j]*x[j]
        x[i]=(b[i]-sumOffDiag)/A[i,i]
        #x[i]=oldx[i]+1.*(x[i]-oldx[i])
    error=sum([(x[i]-oldx[i])**2 for i in range(4)])
timechng=time.time()-start
print 'g-s ', timechng

#raw_input('enter')
x=N.zeros([4],'f')
error=1.
start=time.time()
while error>1.e-9:
    oldx[:]=x[:]
    for i in range(4):
        sumOffDiag=0
        for j in range(4):
            if i!=j:
                sumOffDiag=A[i,j]*x[j]
        x[i]=(b[i]-sumOffDiag)/A[i,i]
        x[i]=oldx[i]+1.25*(x[i]-oldx[i])
    error=sum([(x[i]-oldx[i])**2 for i in range(4)])
    #print error,x[i]
timechng=time.time()-start
print 'sor ', timechng


#using numpy


InvD=(1./A.diagonal())*N.identity(4)
offD=A-A.diagonal()*N.identity(4)
x=N.zeros([4])
xold=x.copy()

start=time.time()
while error>1.e-9:
    x=N.dot(invD,(b-N.dot(offD,xold)))
    xold=x.copy()
    error=(N.dot(x,x.conj()))
timechng=time.time()-start
print 'numpy Jacob', timechng
```

Nonlinear Equations

Quadratic Equations

Second order polynomial equations can be rearranged to find the roots of the equation.

$$y = ax^2 + bx + c$$
$$0 = ax^2 + bx + c$$

To solve this the square must be completed and factored.

$$x^2 + \frac{b}{a}x = -\frac{c}{a}$$
$$x^2 + \frac{b}{a}x + \frac{b^2}{4a^2} = -\frac{c}{a} + \frac{b^2}{4a^2}$$
$$\left(x + \frac{b}{a}\right)^2 = -\frac{c}{a} + \frac{b^2}{4a^2}$$
$$\pm\left(x + \frac{b}{2a}\right) = \sqrt{-\frac{c}{a} + \frac{b^2}{4a^2}}$$
$$x = \pm\sqrt{-\frac{c}{2a} + \frac{b^2}{4a^2}} - \frac{b}{2a}$$
$$x = \pm\sqrt{-\frac{c}{2a} + \frac{b^2}{4a^2}} - \frac{b}{2a}$$
$$x = \pm\sqrt{-\frac{4ac}{4a^2} + \frac{b^2}{4a^2}} - \frac{b}{2a}$$

Identifying Regions with Roots

- roots are 'zeros' of equation
- recast equation so all terms on one side, solve for unknown when function is zero.

$$x^2 = 4$$
$$f(x) = x^2 - 4$$

- plotting function to estimate where it crosses x-axis
- identifying where sign of function changes to bracket regions
  - need to use caution
  - may bracket multiple roots
  - sign change may identify singularities (eg. tangent function)

Bisection Method

- find region with single root
  - product of function values negative

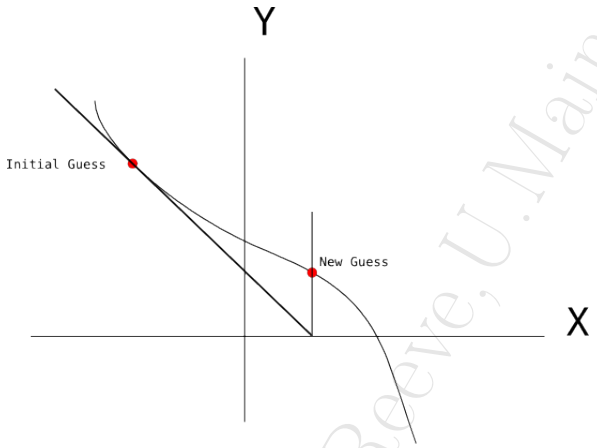
$$f(x_{p1}) \cdot f(x_{p2}) < 0$$

  - divide region in half and finding which zone brackets root
  - repeat until satisfied
  - slow but robust method

Newton's Method

General method for finding a root of equation.

- Make guess, calculate point (x,y)
- calculate derivative, can be estimated using finite-difference approximations
- use initial guess and derivative to project back to x axis (y=0)
- use x value at x-axis to make new guess
- repeat until 'close enough' to zero



Use Newton's method to solve:

$$0 = x^3 - 10$$

X	f(x)	F'(x) Anlytc	f(x-dx)	f(x+dx)	F'(x) Nmrc	New X Anlytc	Resid.
4	54	48	53.04	54.96	48	2.88	13.76
2.88	13.76	24.8	13.27	14.26	24.8	2.32	2.49
2.32	2.49	16.15	2.17	2.81	16.15	2.17	0.16
2.17	0.16	14.07	-0.12	0.45	14.07	2.15	0

$\Delta X = .02$ for numerical estimate of derivative. Each line shows the calculations to update the estimate for X.
Newton's method is based on a truncated Taylor series.

$$f(x_{new}) = f(x_{old}) + (x_{new} - x_{old})\frac{d(f(x))}{dx}$$
$$0 = f(x_{old}) + (x_{new} - x_{old})\frac{d(f(x))}{dx}$$
$$\frac{-f(x_{old})}{\frac{d(f(x))}{dx}} = x_{new} - x_{old}$$
$$x_{old} - \frac{f(x_{old})}{\frac{d(f(x))}{dx}} = x_{new}$$

Newton-Raphson

- Newton's method can be expanded to a series of simultaneous equations
- used to solve simultaneous non-linear equations
- based on multidimensional form of the truncated Taylor Series:

$$f(x + \Delta x, y + \Delta y) = f(x, y) + \Delta x\frac{\partial f(x, y)}{\partial x} + \Delta y\frac{\partial f(x, y)}{\partial y}$$

A similar equation could be written for any number of unknowns.

Secant Method

- similar to Newton Method
- search line based on two initial guesses
- new 'guess' based on x-value that forces f(x)=0 along secant line

$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{f(x_3) - f(x_1)}{x_3 - x_1}$$
$$\frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{0 - f(x_1)}{x_3 - x_1}$$
$$x_3 = x_1 - \frac{x_2 - x_1}{f(x_2) - f(x_1)} \cdot f(x_1)$$

- calculate $f(x_3)$
- select new new point and old point closest to zero $(\min(|f(x_1)|, |f(x_2)|))$
- repeat until close enough

Secant Method