APPM 4600 Lab 14

5 December 2024

The code for this lab can be seen at the end of this document, or on github here.

1 Solving Square Systems

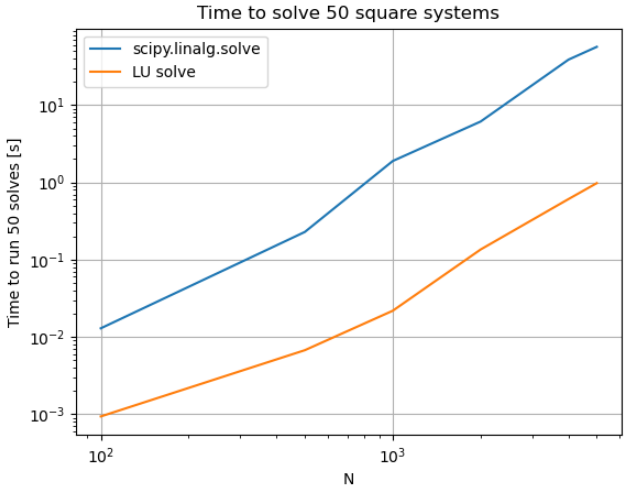
The time taken for the two solution techniques with different size matrices is given in the table below.

N	`scipy.linalg.solve` [ms]	LU factorization [ms]	LU Solve [ms]
100	2.557516098022461	0.1647472381591797	0.0388622839355469
500	13.90385627746582	2.142190933227539	0.10395050048828125
1000	19.914865493774414	10.042428970336914	0.3666877746582031
2000	80.60383796691895	43.37739944458008	2.206087112426758
4000	489.3050193786621	353.6355495452881	6.650209426879883
5000	789.3681526184082	638.4170055389404	9.52768325805664

We find that LU factorization and solving is always faster than using the built in solve function, which is unexpected. This may be an artifact from how timing was measured.

The table below gives the time taken to solve 50 versions of the system $Ax = b$, where A remains fixed and b is chosen randomly each round. The LU solution has a clear advantage, since the expensive part of the algorithm, the LU factorization $O(n^3)$, only has to run once while the triangular solve is cheaper $O(n^2)$.

N	`scipy.linalg.solve` [s]	LU factorization [s]
100	0.012948989868164062	0.0009374618530273438
500	0.22867417335510254	0.0067484378814697266
1000	1.8918616771697998	0.021845102310180664
2000	6.13384485244751	0.13550925254821777
4000	38.78315711021423	0.6089046001434326
5000	56.77767086029053	0.9785189628601074



2 Solving Over Determined Systems

We wish to find the least squares solution to the system $A\mathbf{x} = \mathbf{b}$, where A is non square and the system is overdetermined. One approach is to solve the normal equation,

$$A^T A\mathbf{x} = A^T\mathbf{b}.$$

Another approach is to take the QR factorization of A, $A = QR$, then multiplying by Q^T we have

$$Q^T QR\mathbf{x} = Q^T\mathbf{b}.$$

Since Q is orthogonal, this becomes

$$R\mathbf{x} = Q^T\mathbf{b},$$

where the fact that R is upper triangular may make this faster to solve than the normal equation.

Both techniques for the least squares solution were implemented. Using an error defined as

$$e = |A\mathbf{x} - b|_2,$$

the QR algorithm generally achieves a lower error than solving the normal equation. For example, a set of random matrices generated with the code provided yields $e_{normal} = 0.73, e_{QR} = 0.62$.

As the smallest entry of $\mathbf{d}$ is increased and A becomes increasingly poorly conditioned, the numerical error increases.

```
#! /usr/bin/env python3

import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as la
import scipy.linalg as scila
import time


def question3_2_2(N):

    ''' create  matrix for testing different ways of solving a square
    linear system'''

    '''' N = size of system '''
    print(str(N) + " & ", end="")
    ''' Right hand side '''
    b = np.random.rand(N,1)
    A = np.random.rand(N,N)


    t = time.time()
    x1 = scila.solve(A,b)
    e = time.time()
    print(str((e-t)*1000) + " & ", end="")

    t = time.time()
    lu, piv = scila.lu_factor(A)
    e = time.time()
    print(str((e-t)*1000) + " & ", end="")

    t = time.time()
    x2 = scila.lu_solve((lu, piv), b)
    e = time.time()
    print(str((e-t)*1000) + " \\\\")

# test many right hand side solves
def question3_2_3(N):
    A = np.random.rand(N, N)

    print(str(N) + " & ", end="")

    t = time.time()
```

```
    for i in range(50):
        b = np.random.rand(N, 1)
        x1 = scila.solve(A, b)
    e = time.time()
    print(str(e-t) + " & ", end="")

    t = time.time()
    lu, piv = scila.lu_factor(A)
    for i in range(50):
        b = np.random.rand(N, 1)
        x2 = scila.lu_solve((lu, piv), b)
    e = time.time()
    print(str(e-t) + " \\\\")

def question3_4_1():
    ''' Create an ill-conditioned rectangular matrix '''
    N = 10
    M = 5
    A = create_rect(N,M)
    b = np.random.rand(N,1)

    # solve via normal equation
    x1 = la.solve(np.matmul(A.T, A), np.dot(A.T, b))
    # solve via QR
    Q, R = la.qr(A)
    x2 = la.solve(R, np.dot(Q.T, b))

    # compute L2 norm of solutions
    err1 = la.norm(np.matmul(A, x1) - b)
    err2 = la.norm(np.matmul(A, x2) - b)

    print(err1)
    print(err2)



def create_rect(N,M):
    ''' this subroutine creates an ill-conditioned rectangular matrix'''
    a = np.linspace(1,15,M)
    d = 10**(-a)

    D2 = np.zeros((N,M))
    for j in range(0,M):
       D2[j,j] = d[j]

    '''' create matrices needed to manufacture the low rank matrix'''
    A = np.random.rand(N,N)
    Q1, R = la.qr(A)
    test = np.matmul(Q1,R)
    A =    np.random.rand(M,M)
    Q2,R = la.qr(A)
    test = np.matmul(Q2,R)

    B = np.matmul(Q1,D2)
    B = np.matmul(B,Q2)
    return B


if __name__ == '__main__':
```

```
# run the drivers only if this is called from the command line
question3_4_1()
#for N in [100, 500, 1000, 2000, 4000, 5000]:
#       question3_2_2(N)
#       question3_2_3(N)
```