Becs-114.1100 Computational Science – exercise round 5

Kunal Ghosh, 546247

October 27, 2015

1 Solution to Question 2

1.1 Solving linear equations using Scaled Pivoting. Weights from corresponding hilbert matrices

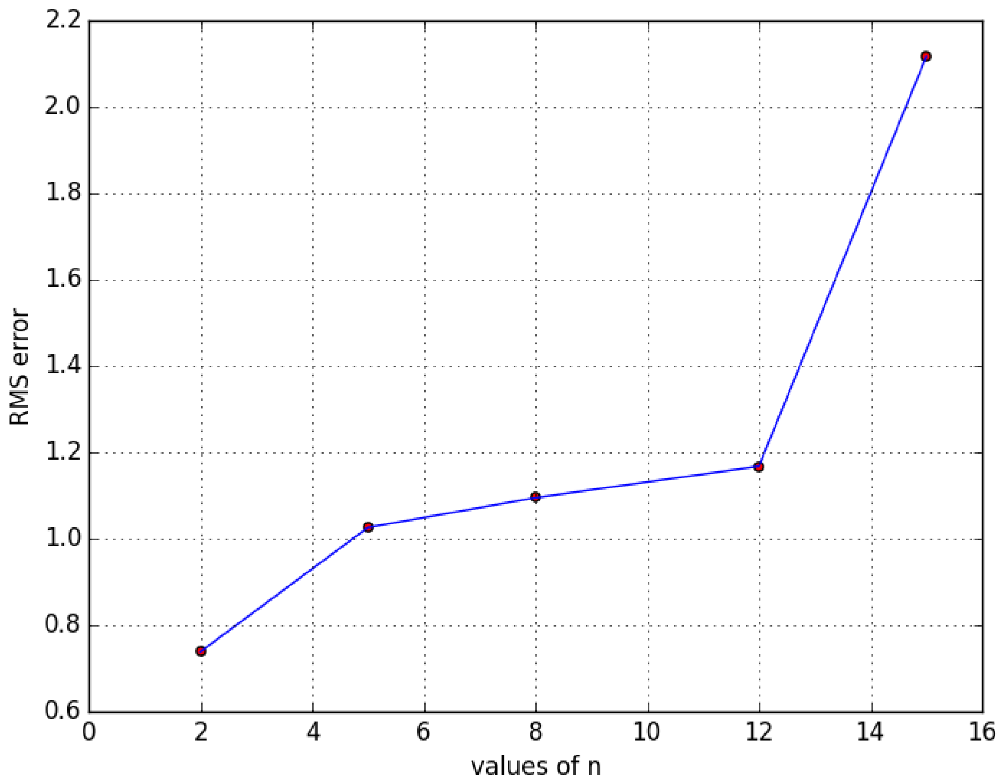

Figure 1: Plot showing that as n increases the RMS error of the solution of a system of linear equations, with the hilbert matrix values as its coefficients, increases.The red dots show the actual error values

The table of n and errors appears on page 4. The corresponding python code can be found at 2

1.2 Solving system of linear equations with coefficients as hilbert(n) for n=3

Solving a system of linear equations with weights as the hilbert matrix.
Calculations performed upto 3 decimal places.

$$A = \text{hilbert}(3) \begin{bmatrix} 1 & 0.5 & 0.333 \\ 0.5 & 0.333 & 0.25 \\ 0.333 & 0.25 & 0.2 \end{bmatrix} \qquad b = \begin{bmatrix} 1.833 \\ 1.083 \\ 0.783 \end{bmatrix}$$

$$I = [1 \quad 2 \quad 3] \qquad \text{Scale Vector} \quad S = \begin{bmatrix} 1.0 \\ 0.5 \\ 0.333 \end{bmatrix}$$

Iteration 1: $\frac{|A_{i1}|}{S_i}$ $i = 1, 2, 3$ $= \begin{bmatrix} 1/1 \\ 0.5/0.5 \\ 0.333/0.333 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ $\therefore$ first highest $= 1 =$ pivot

- swap $(I[1], I(\text{pivot}]) = \text{swap}(I[1], I[1]) \Rightarrow I = [1, 2, 3]$
- Find Multiplier for Row 2. $= A[\text{row} = 2, \text{column} = 1] / A[\text{pivot}, \text{column} 1]$

$$= 0.5/1 = 0.5.$$

Multiply pivot row by 0.5 & subtract from 2.

- Similarly for Row 3 $= A[\text{row} = 3, \text{column} = 1] / A[\text{pivot}, \text{column} 1]$

$$= 0.333/1 = 0.333.$$

Multiply pivot row by 0.333 & subtract from 3.

$$= \left[\begin{array}{ccc|c} 1 & 0.5 & 0.333 & 1.833 \\ 0 & 0.333 - 0.25 & 0.25 - 0.333 \times 0.5 & 1.083 - 1.833 \times 0.5 \\ 0 & 0.25 - 0.25 \times 0.333 & 0.2 - 0.333 \times 0.333 & 0.783 - 1.833 \times 0.333 \end{array}\right]$$

$$= \left[\begin{array}{ccc|c} 1 & 0.5 & 0.333 & 1.833 \\ 0 & 0.083 & 0.083 & 0.166 \\ 0 & 0.083 & 0.089 & 0.173 \end{array}\right]$$

Iteration 2: $\frac{|A_{i2}|}{S_i}$ $i = 2,3$ $= \begin{bmatrix} 0.083/0.5 \\ 0.083/0.333 \end{bmatrix} = \begin{bmatrix} 0.166 \\ 0.249 \end{bmatrix}$ pivot = row 3.

swap $(I[2], I[3]) \Rightarrow I = [1, 3, 2]$

- Find Multiplier for Row 2 = A[Row = 2, column = 2] / A[Pivot, col = 2]

$= 0.083 / 0.083 = 1.$

- Multiply pivot row by multiplier & subtract from 2.

$$\left[\begin{array}{ccc|c} 1 & 0.5 & 0.333 & 1.833 \\ 0 & \cancel{0.083}\ 0 & -0.006 & -0.007 \\ 0 & 0.083 & 0.089 & 0.173 \end{array}\right]$$

<u>Back Substitution</u>: reverse order of $I = [1, 3, 2]$.

row 2 ; $x_3 = -0.007 / -0.006 = 1.167$

row 3 ; $x_2 = [0.173 - (1.167 \times 0.089)] / 0.083. = 0.833$

row 1 ; $x_1 = [1.833 - (1.167 \times 0.333 + 0.833 \times 0.5)] / 1. = 1.028$

$\therefore X = \begin{bmatrix} 1.028 \\ 0.833 \\ 1.167 \end{bmatrix}$

Condition number of hilbert (3) = 524.056.

From the thumb rule: if Condition number is 10^k then we may loose upto k digits of accuracy. Over and above the Numerical loss.

And we can see here that our results are off by 2 digits of precision (A little more than 2 but that can be attributed to the numerical error.).

$\therefore$ Because of the high ~~hibert~~ Condition number. The precision we are loosing by Keeping upto 3 decimal places is causing a ~~har~~ Large change in the output. over and above the numerical error.

n	RMS Error	Condition Number
2	0.73833521	15.0167409881
5	1.02602254	282901.77002
8	1.09505307	7657562245.89
12	1.16783884	5.89342127254e+15
15	2.11656192	1.74359790918e+17

Table 1: Table showing the values of n and the RMS error after solving the system of linear equations with **hilbert(n)** as the coefficients.

2 Appendix A

Python source code for 1.1.

```
from __future__ import division
from scipy.linalg import hilbert
import numpy as np
from pprint import pprint
import pylab as pl

def new_solve(A,b):
    A = np.asarray(A, np.float)
    b = np.asarray(b, np.float)
    # Create the scale vector max(abs(ri)) of each row ri in A
    S = np.zeros(A.shape[0], np.float)
    S = np.max(np.abs(A), axis=1)
    # Create the Index vector
    I = np.asarray(range(np.max(S.shape)))
    # iterate over as many times as there are rows (r = 0 to rmax)
    for idx in range(len(I)):
        r = I[idx]
        # get the values from the idx(th) column
        rows_we_want = I[idx:]
        corresponding_column_values = A[rows_we_want, idx]
        # divide the column values by their corresponding scale and get the index of the row with max value
        div_val = np.true_divide(np.abs(corresponding_column_values),S[rows_we_want])
        I_idx = np.argmax(div_val)
        # because the above index is in I, the actual row is
        act_idx = I[idx:][I_idx]
        max_row = A[act_idx,:]
        # swap current Idx with max_row's idx
        # swap the 0th idx and the new max in the sub array of I
        I[idx:][0], I[idx:][I_idx] = I[idx:][I_idx], I[idx:][0]
        # iterate over remaining rows and update them
        for rem_rows in I[idx+1:]:
            # Get the appropriate multiple of the pivot row. to make the remaining row's idx column a zero
            multiplier = np.true_divide(A[rem_rows][idx],max_row[idx])
            A[rem_rows,idx:] -= max_row[idx:] * multiplier
            b[rem_rows] -= b[act_idx] * multiplier
    return I, A, b

def gauss(I, A, b):
    # returns the solutions to x
    x = np.zeros(I.shape)
    # because this is directly used in indexing and
    # max index of x would be len(x) -1
    len_x = len(x)-1
    # reverse I because we go in reverse I order.
    I = I[::-1]
    for count,row in enumerate(I):
        # get the row which we need to evaluate.
        weighted_sum_of_already_computed_x = 0
        for i in range(count):
            # if its the first value, we need to evaluate once.
            # for the second value, we need to evaluate twice and so on.
            col = len_x-i
            weighted_sum_of_already_computed_x += A[row, col] * x[col]
        # len(x)-count-1 because indices from 3 to 0 when len(x) = 4
        x[len_x-count] = (b[row] - weighted_sum_of_already_computed_x) / A[row,len_x-count]
    return x

def error(x,x_actual):
    diff = np.abs(x-x_actual)
    return np.sqrt(np.divide(np.sum(diff ** 2),x.shape))


if __name__ == '__main__':
    errs = []
    errors = []
    n_vals = [2,5,8,12,15]
    for n in n_vals:
        A = hilbert(n)
        b = np.sum(A,axis=1)
        I,A,b = new_solve(A,b)
        x = gauss(I,A,b)
        errors.append(error(x,b))
        print n,errors[-1]
    pl.plot(n_vals, errors,c='b')
    pl.grid()
    pl.scatter(n_vals, errors,c='r',marker="o")
    pl.xlabel("values of n")
    pl.ylabel("RMS error of partial pivot and actual solution of hilbert(n)")
    pl.show()
```