Numerical Computing :: Project Five

Julia Troni

In [1]:

```
%matplotlib notebook
from matplotlib import pyplot
import matplotlib.pyplot as plt
import numpy as np
import math
import timeit
```

In [2]:

```
#import linalg package of the SciPy module for the LU decomp
import scipy.linalg as linalg
```

Numerical Experiements, Method Implementation, and Data Visualization

In [3]:

```
#1. Generate a right-hand-side b of all ones of appropriate size.
def generate_bs(matrix):
    # https://numpy.org/doc/stable/reference/generated/numpy.ones.html
    ##returns array of 1s with same n dimension at matrix
    return np.ones((matrix.shape[0], 1), dtype=type(matrix[0][0]))
```

In [4]:

```
#2. Solve Ax = b with a generic linear solver. Call the resulting vector truth

def solveTruth(matrix):
    b= generate_bs(matrix)
    return np.linalg.solve(matrix, b)
```

In [5]:

```
#3.Solve Ax=b with LU decomposition or the Cholesky factorization, depending on whether

def isSymmetric(mat):
    #3. Is it symmetric?
    symm=False
    ##first transpose
    trans= mat.transpose()
    # now compare matrices using array_equal() method
    if np.array_equal(trans, mat):
        symm=True

    return symm;

def LU_solve(matrix):
    b = generate_bs(matrix)
    #call the lu_factor function
    LU = linalg.lu_factor(matrix)
    #solve given LU and b
    x = linalg.lu_solve(LU, b)
    return x
```

```python
def choleskyDecomp(matrix):
    #Cholesky decomposition with scipy
    #https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.cho_solve.html
    c, low = linalg.cho_factor(matrix)
    x = linalg.cho_solve((c, low), generate_bs(matrix))
    return x
```

In [6]:

```python
#4. Solves Ax = b using the Jacobi method
def jacobi(A, max_iters=25, x=None):
    b=generate_bs(A)
    # Create an initial guess
    x = generate_bs(A)
    # Create a vector of the diagonal elements of A and subtract them from A
    D = np.diag(A)
    R = A - np.diagflat(D)
    try:
        # Iterate for max_iters times
        for i in range(max_iters):
            temp = x
            x = (b - np.dot(R,x)) / D
    except np.linalg.LinAlgError:
        return temp
    return x
```

In [7]:

```python
#5. solves Ax = b using the Gauss-Seidel method

def gauss_seidel(A, num_iters=25):

    b=generate_bs(A)
    # Create an initial guess
    x = np.ones((A.shape[0], 1))
    L = np.tril(A)
    U = A - L
    try:
        # Iterate for num_iters times
        for i in range(num_iters):
            temp = x
            x = np.dot(np.linalg.inv(L), b - np.dot(U, x))
    except np.linalg.LinAlgError:
        return temp
    return x
```

For the timing studies I ran the following code snippet for each matrix. From this we can see the trend that as each matrix grows in dimensions, the execution time increases significantly. We also see that the gauss_seidel method is about 2x as costly in time compared to the jacobi method

```
>> %%timeit
>> jacobi(matrix)

matrix1: 431 µs ± 24.3 µs per loop (mean ± std. dev. of 7 runs, 1000
loops each)
matrix2: 690 µs ± 44.7 µs per loop (mean ± std. dev. of 7 runs, 1000
loops each)
```

```
matrix4: 1.49 ms ± 32.9 µs per loop (mean ± std. dev. of 7 runs, 1000
loops each)
matrix5: 494 ms ± 25.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

```
>> %%timeit
>> gauss_seidel(matrix)

matrix1: 1.48 ms ± 76.3 µs per loop (mean ± std. dev. of 7 runs, 1000
loops each)
matrix2: 1.97 ms ± 141 µs per loop (mean ± std. dev. of 7 runs, 1000
loops each)
matrix4: 3.84 ms ± 497 µs per loop (mean ± std. dev. of 7 runs, 100 loops
each)
matrix5: 832 ms ± 45.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

In [8]:

```
def relative_error(truth, sol): #truth is from
    return np.linalg.norm(sol - truth) / np.linalg.norm(sol)
```

In [9]:

```
mat1= np.loadtxt('mat1.txt',dtype=float, encoding=None, delimiter=",")
mat2= np.loadtxt('mat2.txt',dtype=float, encoding=None, delimiter=",")
mat3= np.loadtxt('mat3.txt',dtype=float, encoding=None, delimiter=",")
mat4= np.loadtxt('mat4.txt',dtype=float, encoding=None, delimiter=",")
mat5= np.loadtxt('mat5.txt',dtype=float, encoding=None, delimiter=",")
```

In [10]:

```
def analyze():
    for i in [mat1,mat2,mat4,mat5]: #matrix 3 give me (more) problems and I took my ang

        ##hacky solution for printing the name of the matrix
        if len(i)==len(mat1):
            print("*********Matrix 1***********")
        elif len(i)==len(mat2):
            print("*********Matrix 2***********")
        elif len(i)==len(mat4):
            print("*********Matrix 4***********")
        elif len(i)==len(mat5):
            print("*********Matrix 5***********")

        #"truth" value to compare with relative error
        truth=solveTruth(i)

        ##analyzing the relative errors of each method with each matrix
        if isSymmetric(i):
            q3=choleskyDecomp(i)
            print("Symmetic matrix using Colesky method: relative error: ", relative_er
        else:
            q3=LU_solve(i)
            print("Nonsymmetric matrix using LU decomposition: relative error: ",relati

        jacob=jacobi(i)
        print("Jacobi method relative error: ", relative_error(jacob, truth))
```

```
        seidel=gauss_seidel(i)
        print("Seidel method relative error: ", relative_error(seidel, truth))

analyze()
```

```
*********Matrix 1***********
Nonsymmetric matrix using LU decomposition: relative error:  0.0
Jacobi method relative error:  1096.552629701177
Seidel method relative error:  0.0
*********Matrix 2***********
Symmetic matrix using Colesky method: relative error:  4.51782998967664e-15
Jacobi method relative error:  277332918.06574994
Seidel method relative error:  0.18274588441512818
*********Matrix 4***********
Nonsymmetric matrix using LU decomposition: relative error:  0.0
Jacobi method relative error:  7.888156998721891e+34
Seidel method relative error:  1.9250057937425232e+36
*********Matrix 5***********
Symmetic matrix using Colesky method: relative error:  1.3470560606988453e-15
Jacobi method relative error:  20.132514553823146
Seidel method relative error:  0.6719100931357497
```

Analysis

The relative error in the jacobi method appears to be significantly larger than the linalg.solve "truth"

Jacobi method is an iterative method that only converges for any initial guess if the matrix is strictly row diagonally dominant. When the diagonal elements are dominant this ensures the iterative methods converge to a solution, otherwise the solution may not converge at all.

So from this error comparison, I assume that these matrices are NOT diagonally dominant, and hence do not converge using the Jacobi method

On the other hand the Gauss-Seidel method can be applied to more matrices since it converges for any initial guess if the matrix is strictly diagonally dominant, or if the matrix is symmetric positive definite. So, from aboce it appears that these matrices are symmetric positive definite, but NOT diagonally dominant

References

- As usual my go to resource https://pythonnumericalmethods.berkeley.edu/notebooks/chapter14.05-Solve-Systems-of-Linear-Equations-in-Python.html

- Lectures were quite helpful (and especially the 5.2 with the short clip of "Numerical Computation brought to you by...". That made me laugh, thank you)

- https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_IterativeSolvers.html for a majority of the implementations

- http://www.math.iit.edu/~fass/477577_Chapter_13.pdf

In [ ]: