HW5

October 9, 2020

0.1 ### HW 5

Andrew Pickner

I worked by myself on this assignment.

---

[1]:
```
import numpy as np
from scipy.io import mmread
import timeit
import warnings

# I was getting a RuntimeWarning: overflow encountered in true_divide in␣
↪jacobi's method
warnings.filterwarnings('ignore')
```

[2]:
```
'''
read_matrix
    file_name: file where the matrix resides

Reads the matrix in from memory using numpy
'''
def read_matrix(file_name):
    # https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html
    ext = file_name[-3:]
    if ext == "txt":
        return np.loadtxt(file_name,delimiter=',')
    elif ext == "mtx":
        return mmread(file_name).A
```

[3]:
```
'''
Question #1 functions:

generate_ones
    matrix: a `n x m` matrix stored in a np.ndarray
    ------
    RETURNS: a `n x 1` np.ndarray of all ones using numpy
'''
```

```
def generate_ones(matrix):
    # https://numpy.org/doc/stable/reference/generated/numpy.ones.html
    return np.ones((matrix.shape[0], 1), dtype=type(matrix[0][0]))
```

[4]:
```
'''
Question #3 functions:

lu
    A: an `n x m` matrix stored in a np.ndarray
    ------
    RETURNS: a tuple containing the L and U matrices obtained from performing␣
 ↪LU decomp.

lu_solve
    A: an `n x m` matrix stored in a np.ndarray
    b: an `n x 1` matrix stored in a np.ndarray
    ------
    RETURNS: an `n x 1` matrix stored in a np.ndarray containing a solution x␣
 ↪to the equation Ax=b

cholesky
    A: an `n x m` matrix stored in a np.ndarray
    ------
    RETURNS: the L matrix obtained from performing cholesky decomp.

cholesky_solve
    A: an `n x m` matrix stored in a np.ndarray
    b: an `n x 1` matrix stored in a np.ndarray
    ------
    RETURNS: an `n x 1` matrix stored in a np.ndarray containing a solution x␣
 ↪to the equation Ax=b

AS FOR:
    forward_sub
    back_sub

    Although I roughly understood what these algorithms were doing going into␣
 ↪the assignment,
    https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_LU.
 ↪html was incredibly
    useful because I didn't want to spend considerable time implementing these␣
 ↪functions because we could've
    simply used a library function to solve.
'''
def p3_solve(A,b):
    a = np.array(A, dtype=np.float128)
```

```
    if is_symmetric(a):
        return cholesky_solve(a,b)
    return lu_solve(a,b)

def lu(A):
    #  n = number of rows
    n = A.shape[0]
    U = A.copy()
    L = np.eye(n, dtype=np.double)
    #Loop over rows
    for i in range(n):
        #Eliminate entries below i with row operations
        #on U and reverse the row operations to
        #manipulate L
        factor = U[i+1:, i] / U[i, i]
        L[i+1:, i] = factor
        U[i+1:] -= factor[:, np.newaxis] * U[i]
    return L, U

def lu_solve(A, b):
    L, U = lu(A)
    # solve Ly=b for y using forward sub
    y = forward_sub(L, b)
    # then solve Ux=y for x using back sub
    return back_sub(U, y)
    # seemed rather consistent with the scipy method so I kept the forward and␣
 ↪backward that I have.
    # return solve_triangular(U, b)

def cholesky(A):
    #  n = number of rows
    n = A.shape[0]
    L = np.zeros((n, n), dtype=np.double)
    for k in range(n):
        L[k, k] = np.sqrt(A[k, k] - np.sum(L[k, :] ** 2))
        L[(k+1):, k] = (A[(k+1):, k] - L[(k+1):, :] @ L[:, k]) / L[k, k]
    return L

def cholesky_solve(A, b):
    L = cholesky(A)
    # just special case of LU decomp where U = L.T
    y = forward_sub(L, b)
    return back_sub(L.T, y)

def forward_sub(L, b):
    #  n = number of rows
    n = L.shape[0]
```

```
    # allocating space for the solution vector
    y = np.zeros_like(b, dtype=np.double);
    #Here we perform the forward-substitution.
    #Initializing  with the first row.
    y[0] = b[0] / L[0, 0]
    #Looping over rows in reverse (from the bottom  up),
    #starting with the second to last row, because  the
    #last row solve was completed in the last step.
    for i in range(1, n):
        y[i] = (b[i] - np.dot(L[i,:i], y[:i])) / L[i,i]
    return y

def back_sub(U, y):
    #  n = number of rows
    n = U.shape[0]
    # allocating space for the solution vector
    x = np.zeros_like(y, dtype=np.double);
    #Here we perform the back-substitution.
    #Initializing with the last row.
    x[-1] = y[-1] / U[-1, -1]
    #Looping over rows in reverse (from the bottom up),
    #starting with the second to last row, because the
    #last row solve was completed in the last step.
    for i in range(n-2, -1, -1):
        x[i] = (y[i] - np.dot(U[i,i:], x[i:])) / U[i,i]
    return x
```

[5]:
```
'''
Question #4 functions:

jacobi_solve
    A: an `n x m` matrix stored in a np.ndarray
    b: an `n x 1` matrix stored in a np.ndarray
    max_iters=25: can change the number of max iterations the method will use,␣
 ↪25 by default
    x=None: can supply an inital guess for the solution x, I initialize x to be␣
 ↪all ones based on A when x==None
    ------
    RETURNS: an `n x 1` matrix stored in a np.ndarray containing a solution x␣
 ↪to the equation Ax=b using
             jacobi's iterative method
'''
def jacobi_solve(A, b, max_iters=25, x=None):
    """Solves the equation Ax=b via the Jacobi iterative method."""
    # Create an initial guess if needed                                        ␣
 ↪                                                                            ␣
 ↪
```

```
    if x is None:
        x = generate_ones(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
    # p5 giving me problems
    except np.linalg.LinAlgError:
        return temp
    return x
```

[6]:
```
'''
Question #5 functions:

gauss_seidel_solve
    A: an `n x m` matrix stored in a np.ndarray
    b: an `n x 1` matrix stored in a np.ndarray
    max_iters=25: can change the number of max iterations the method will use, 25 by default
    x=None: can supply an inital guess for the solution x, I initialize x to be all ones based on A when x==None
    ------
    RETURNS: an `n x 1` matrix stored in a np.ndarray containing a solution x to the equation Ax=b using
             gauss/seidel's iterative method
'''
def gauss_seidel_solve(A, b, num_iters=25, x=None):
    # Create an initial guess if needed
    if x is None:
        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))
    # p5 giving me problems
    except np.linalg.LinAlgError:
        return temp
    return x
```

[7]:
```
'''

OTHER RELEVANT HELPER FUNCTIONS:

    relative_error
        truth: `n x m` matrix that is claiming to be the absolute correct values
        sol: `n x m` matrix that is any arbitrary solution obtained
        ------
        RETURNS: the relative error of the arbitrary solution to the true␣
 ↪solution

    is_symmetric
        matrix: `n x m` matrix
        ------
        RETURNS: true if the matrix is symmetric, and false otherwise

'''
def relative_error(truth, sol):
    return np.linalg.norm(sol - truth) / np.linalg.norm(sol)

def is_symmetric(matrix):
    if matrix.shape[0] == matrix.shape[1]:
        if isinstance(matrix[0][0], float):
            return np.allclose(matrix,matrix.T, atol=1e-06)
        return np.array_equal(matrix,matrix.T)
    return False
```

[8]:
```
'''

TIMING MODULE:
https://www.pythoncentral.io/time-a-python-function/

'''
def wrapper(func, *args, **kwargs):
    def wrapped():
        return func(*args, **kwargs)
    return wrapped
```

[9]:
```
num_matrices = 4
num_tests = 50

directory = "/Users/AndrewMacbook/Downloads/"

for i in range(num_matrices):
    file_name = "{}mat{}-1.txt".format(directory, i+1)
    matrix = read_matrix(file_name)
    b = generate_ones(matrix)

    truth = np.linalg.solve(matrix, b)

    rel_error_tru = relative_error(truth, truth)
    rel_error_lu  = relative_error(truth, p3_solve(matrix, b))
    rel_error_jac = relative_error(truth, jacobi_solve(matrix, b))
    rel_error_gs  = relative_error(truth, gauss_seidel_solve(matrix, b))

    linalg_solved      = wrapper(np.linalg.solve, matrix, b)
    luCholesky_solved  = wrapper(p3_solve, matrix, b)
    jacobi_solved      = wrapper(jacobi_solve, matrix, b)
    gaussSeidel_solved = wrapper(gauss_seidel_solve, matrix, b)
    print("{} Matrix {} {}".format("#" * 12, i+1, "#" * 12))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("linsol",timeit.
 ↪timeit(linalg_solved, number=num_tests) / num_tests, rel_error_tru))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("LU/Chol",timeit.
 ↪timeit(luCholesky_solved, number=num_tests) / num_tests, rel_error_lu))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("Jacobi",timeit.
 ↪timeit(jacobi_solved, number=num_tests) / num_tests, rel_error_jac))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("GS",timeit.
 ↪timeit(gaussSeidel_solved, number=num_tests) / num_tests, rel_error_gs))
```

```
############ Matrix 1 ############
linsol   time: 1.087941e-03, error: 0.000000e+00
LU/Chol  time: 2.189109e-02, error: 3.671191e-01
Jacobi   time: 2.531968e-02, error: 1.704764e-04
GS       time: 8.308061e-02, error: 1.500389e-06
############ Matrix 2 ############
linsol   time: 3.455016e-03, error: 0.000000e+00
LU/Chol  time: 3.668489e-02, error: 2.188609e-02
Jacobi   time: 9.041695e-02, error: 1.177007e-12
GS       time: 2.664884e-01, error: 2.635684e-09
############ Matrix 3 ############
linsol   time: 9.452872e-04, error: 0.000000e+00
LU/Chol  time: 5.007002e-02, error: 1.002810e-05
Jacobi   time: 2.076657e-02, error: nan
GS       time: 1.262019e-03, error: 2.126029e+00
############ Matrix 4 ############
```

```
linsol   time: 6.492302e-05, error: 0.000000e+00
LU/Chol  time: 3.049418e-03, error: 5.046800e-01
Jacobi   time: 8.678296e-04, error: 9.999775e-01
GS       time: 3.733116e-03, error: 1.355951e-02
```

0.1.1 EXTRA CREDIT:

1. dwa512: Square Dielectric Waveguide

Not entirely sure what that^ means, but from the 'structure plot', it looked like a matrix that would be non-singular, and that the iterative methods would shine. Unfortunately the latter proved to be false.

---

[10]:
```
names = ["dwa512"]

for i in range(len(names)):
    file_name = "{}{}.mtx".format(directory, names[i])
    matrix = read_matrix(file_name)
    b = generate_ones(matrix)

    truth = np.linalg.solve(matrix, b)

    rel_error_tru = relative_error(truth, truth)
    rel_error_lu  = relative_error(truth, p3_solve(matrix, b))
    rel_error_jac = relative_error(truth, jacobi_solve(matrix, b))
    rel_error_gs  = relative_error(truth, gauss_seidel_solve(matrix, b))

    linalg_solved      = wrapper(np.linalg.solve, matrix, b)
    luCholesky_solved  = wrapper(p3_solve, matrix, b)
    jacobi_solved      = wrapper(jacobi_solve, matrix, b)
    gaussSeidel_solved = wrapper(gauss_seidel_solve, matrix, b)

    print("{} {:^20} {}".format("#" * 12, names[i], "#" * 12))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("linsol",timeit.
 ↪timeit(linalg_solved, number=num_tests) / num_tests, rel_error_tru))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("LU/Chol",timeit.
 ↪timeit(luCholesky_solved, number=num_tests) / num_tests, rel_error_lu))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("Jacobi",timeit.
 ↪timeit(jacobi_solved, number=num_tests) / num_tests, rel_error_jac))
    print("{:<8} time: {:.6e}, error: {:.6e}".format("GS",timeit.
 ↪timeit(gaussSeidel_solved, number=num_tests) / num_tests, rel_error_gs))
```

```
############        dwa512        ############
linsol   time: 6.648570e-03, error: 0.000000e+00
LU/Chol  time: 6.648564e-01, error: 3.991839e-12
```

```
Jacobi    time: 1.589301e-01, error: 1.000000e+00
GS        time: 3.930483e-01, error: 1.000000e+00
```

0.1.2 Sources used:

https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_LU.html

https://courses.grainger.illinois.edu/cs357/sp2020/notes/ref-9-linsys.html

http://www.cim.mcgill.ca/~derek/ecsex43/A1.md.html

https://numpy.org

https://www.pythoncentral.io/time-a-python-function/

https://stackoverflow.com/questions/7370801/how-to-measure-elapsed-time-in-python

https://math.nist.gov/MatrixMarket/data/NEP/dwave/dwa512.html

---