CS 111 (S19): Homework 4

Due by 6:00pm Tuesday, April 30

NAME and PERM ID No.: Chen Li, 5468137

UCSB EMAIL: chenli@ucsb.edu

Note: In this homework, you'll be using a few of the functions I introduced in lecture, like LSolve, USolve, the temperature setup, etc. These can all be found in our shared files GitHub repository of "Useful Files" found here (link).

1. The temperature problem models our cabin in the woods in two dimensions, but most modern scientific simulations are done in three dimensions. Here you will create the matrix that corresponds to a 3-D version of the temperature problem. The "cabin" is now the unit cube. As before, we will discretize the interior by dividing it into k points in each dimension, but now there are k^3 points in all rather than k^2. The partial differential equation still leads to the approximation that the temperature at any given point is the average of the temperatures at the neighboring points, but now there are 6 neighbors, with 2 in each dimension.

Using the routine `make_A(k)` from `Temperature.ipynb` as a model, write a routine `make_A_3D(k)` that returns the k^3-by-k^3 matrix A for the 3D version of the temperature problem. This matrix expresses the fact that, in a 3D k-by-k-by-k grid, each interior point has a temperature that is the average of its 6 neighbors (left, right, up, down, in, out). The diagonal elements of A are all equal to 6, and the off-diagonal elements are either 0 or -1. Most of the rows of A have 7 nonzeros.

Here below, for debugging, is the correct matrix for $k = 2$. I converted it to dense for printing—you should also print it out as sparse, and indeed for $k > 2$ it's going to be too large to see what's going on in the dense matrix anyway.

```
[In:]
  k = 2
  A = make_A_3D(k)
  print('k:', k)
  print('dimensions:', A.shape)
  print('nonzeros:', A.size)
  #print('A as sparse matrix:'); print(A)
  print('A as dense matrix:'); print(A.todense())

[Out:]
  k: 2
  dimensions: (8, 8)
  nonzeros: 32
  A as dense matrix:
  [[ 6. -1. -1.  0. -1.  0.  0.  0.]
   [-1.  6.  0. -1.  0. -1.  0.  0.]
   [-1.  0.  6. -1.  0.  0. -1.  0.]
   [ 0. -1. -1.  6.  0.  0.  0. -1.]
   [-1.  0.  0.  0.  6. -1. -1.  0.]
   [ 0. -1.  0.  0. -1.  6.  0. -1.]
   [ 0.  0. -1.  0. -1.  0.  6. -1.]
   [ 0.  0.  0. -1.  0. -1. -1.  6.]]
```

Print out your matrix for $k = 2$ and $k = 3$ as a check that it's correct. Also use `plt.spy(A)` to make a spy plot of the nonzero structure for $k = 4$ or 5 (you may want to zoom in on the plot to see all the structure).

To complete a realistic simulation you would also write a routine `make_b_3D(k)` to compute the right-hand side b. For this problem, you don't have to do that; for the experiments in Problem 2 you can just use `np.random.rand()` to generate a random b.

Note: When you finish, submit your code of `make_A_3D(k)` to Gradescope. The file name must be `make_A_3D.py`. The function name must be `make_A_3D`. A skeleton code file can be found in 04.23 lecture files.

2. Now you will experiment with solving $At = b$ using various solvers from class and from `numpy`. For this problem, you should use the 3-D version of the temperature matrix from Problem 1. (You can get partial credit by using the 2-D temperature matrix from the class instead.) You can use a randomly chosen right-hand side vector b.

Experiment with solving $At = b$ for the temperature t, for various values of k, using five different solvers:

- The `CGsolve()` conjugate gradient solver, from class. (You can vary the arguments `tol` and `max_iters` to make it find a more accurate solution.)
- The `Jsolve()` Jacobi solver, also from class. (Again you can vary `tol` and `max_iters`.)
- The `scipy` sparse conjugate gradient solver `spla.cg()`.
- The `scipy` sparse LU solver `spla.spsolve()`.
- The `LUsolve()` dense LU solver from class. (For this, you will have to use the dense form of A that you get from `A.todense()`. Warning! This will use too much memory if k gets very big at all.)

For each solve, measure the run time and also the relative residual norm. Which solvers are more accurate? Which are faster? How do the answers to these questions change as you change k?

Warning: Start with very small values of k, and be cautious as you increase k! The matrices get big in a hurry. Different solvers will fall over for different values of k; try to see how big a value of k each solver can handle with at most 30 seconds of compute time.

k were set as 5, 10, 20, 30, 40, 50, the tolarance were set to 10^{-15}, and all the max iteration were set base on when the relative residual is smallest to find the executing time accurately.

when k = 5, spla.spsolve is fastest and most accurate, Jsolve and LUsolve are relatively solwer but also accurate engouth. However, CGsolve is the only one can't reach tolerance of 10^{-15}
For spla.cg tolerance are tentative to find the best performance and accuracy because slight if tolerance higher than possible result, it will extremely long time to execute.

when k = 10, spla.cg reach its most accurate point (relres = 3.88e-15) within first 100 iteration but then can't go closer. Therefore, it took more than 4 second to execute just for go through 10000 iterations. so I start to adjust it tolerance. Jsolve go very close to tolerance(relres = 1.09e-15). Jsolve and LUsolve is still very slower

when k = 20, spla.cg is not accurate(relres=7.72e-14)but very fast. Jsolve is accurate(relres=3.79e-15) but slow.CGsolve were set to 200 iteration and reach its maximum accuracy(relres=2.63e-14), which take slightly longer than spla.cg. For LUsolve, it takes too long I decided to interupte it.

when k = 30, spla.cg is still the fastest but not accurate, Jsolve is slow but accurate. Others are in between.

when k = 50, Jsolve take more than 30 seconds, CGsolve this time reaches its accurate place in 1.37 second, but is not accurate(relres=2.42e-13) comparing with spla.cg, which take slightly longer to reach one more decimal place. Only these two left for now.

when k = 70, all solvers need more than 30 second, so I have to adjust tolerance to e-12 to compare. CGsolve is slightly faster and spla.cg is slightly accurate in this case.

here is the structure of the code I use to measure

```
k = 5
b = 10*np.round(np.random.rand(k*k*k))
A = make_A_3D(k)
print(A.todense())
```

```
[[ 6. -1.  0. ...  0.  0.  0.]
 [-1.  6. -1. ...  0.  0.  0.]
 [ 0. -1.  6. ...  0.  0.  0.]
 ...
 [ 0.  0.  0. ...  6. -1.  0.]
 [ 0.  0.  0. ... -1.  6. -1.]
 [ 0.  0.  0. ...  0. -1.  6.]]
```

```
%%time
res = []
def collect_residual(x):
    res.append(npla.norm(A @ x - b) / npla.norm(b))
x, n_iter = spla.cg(A, b, tol=1e-12,callback=collect_residual)
print("cg residuals for each iteration: ")
#for i in range(0, len(res), 100):
#for i in range(len(res)):
    #print("Iter", i+1, "rel_res:", res[i])
```

```
cg residuals for each iteration:
CPU times: user 3.28 ms, sys: 1.5 ms, total: 4.78 ms
Wall time: 3.56 ms
```

```
%%time
X, res = Jsolve(A, b, tol = 1e-12, max_iters = 10000, callback = None)
#for i in range(0, len(res), 100):
#for i in range(len(res)):
    #print("Iter", i+1, "rel_res:", res[i])
```

```
CPU times: user 7.3 ms, sys: 2.4 ms, total: 9.69 ms
Wall time: 7.7 ms
```

```
%%time
X, rel_res = CGsolve(A, b, tol = 1e-12, max_iters = 400, callback = None)
#print("x =", Xcg[0])
#for i in range(0, len(res), 100):
#for i in range(len(rel_res)):
    #print("Iter", i+1, "rel_res:", rel_res[i])
```

```
CPU times: user 2.28 ms, sys: 1.1 ms, total: 3.38 ms
Wall time: 2.32 ms
```

```
%%time
x = spla.spsolve(A, b)
res = npla.norm(A @ x - b) / npla.norm(b)
#print("SPsolve residual :", res)
```

```
CPU times: user 1.32 ms, sys: 1.1 ms, total: 2.42 ms
Wall time: 1.12 ms
```

```
%%time
x, res = LUsolve(np.array(A.todense()), b)
#print("LUsolve rel_res :", res)
```

```
CPU times: user 41.2 ms, sys: 4.22 ms, total: 45.4 ms
Wall time: 42.5 ms
```

here is the result record

k	spla.cg		Jsolve		CGsolve		spla.spsolve		LUsolve	
5	7.67ms	6E-16	51.5ms	9.19E-16	15.7ms	1.17E-15	2.11ms	9.76E-16	40.2ms	1.12E-15
10	4.44s	3.88E-15	4.16s	1.09E-15	61.4ms	5.61E-15	10.2ms	3.52E-15	2.8s	7.97E-15
20	41ms	7.47E-14	5.41	3.79E-15	89.9ms	2.63E-14	376ms	1.74E-14		
30	133ms	7.72E-14	11.2s	8.35E-15	409ms	6.79E-14	3.55s	4.76E-14		
50	1.88s	9.80E-14			1.37s	2.42E-13				
70	2,62s	8.70E-13			2.58s	9.73E-13				

As a conclusion, Jsolve, and spla.spsolve are accurate but slow. CGsolve and cpla.cg are fast but lack of accuracy, which require to adjust to larger tolerance to get use. Since the matrix is SPD, the result for two cg solvers are reasonable.

3. Let

$$A = \begin{pmatrix} 4 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & 4 \end{pmatrix}$$

and let $b = (15, -3, 12)^T$.

3a. Use the `scipy` Cholesky factorization routine `linalg.cholesky()` to compute the triangular Cholesky factor of A. (Either upper or lower triangular is fine, but just compute one of them.) Verify that the answer is correct by multiplying the factor by its transpose and comparing with A. Then use `Usolve()` and/or `Lsolve()` to compute the solution x to $Ax = b$ from the Cholesky factor (without calling any other factorization routine). Show the Jupyter/Python input and output for your computations.

```
A = np.array([[4,-1,-1],[-1,4,-1],[-1,-1,4]])
```

```
L = linalg.cholesky(A)
```

```
b = np.array([15,-3,12])
```

```
x = Usolve(L, Lsolve(L.T, b))
print("x=", x)
print("rel_res=",npla.norm((A@x-b)/b))
```

```
x= [5.4 1.8 4.8]
rel_res= 2.9605947323337506e-16
```

3b. Use the `scipy` QR factorization routine `linalg.qr()` to compute the two matrices (orthogonal and upper triangular) that constitute the QR factorization of A. Verify that the answer is correct by multiplying the factors and comparing with A. Then use `Usolve()` and/or `Lsolve()` to compute the solution x to $Ax = b$ from the QR factors (without calling any other factorization routine). Show the Jupyter/Python input and output for your computations.

$Ax = b \Rightarrow QRx = b \Rightarrow Rx = Q^T b$, then we can use Usolve

```
Q, R = linalg.qr(A)
```

```
print("Residual",npla.norm(Q@R-A))
```

```
Residual 1.7342238036525468e-15
```

```
x = Usolve(R, Q.T @ b)
print("x=", x)
print("rel_res=",npla.norm((A@x-b)/b))
```

```
x= [5.4 1.8 4.8]
rel_res= 3.7914111775644515e-16
```

4. How do you define an orthogonal matrix?
Which of the following matrices are orthogonal? (Don't show your work, just give the answer.)

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad C = \begin{pmatrix} 2 & 0 \\ 0 & 1/2 \end{pmatrix}, \quad D = \begin{pmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix}$$

a matrix Q is a orthogonal matrix means its columns are pair-wise orthonomal with norm of one unit, and its rows are pair-wise orthonomal: $QQ^T = Q^TQ = I$ and $Q^T = Q^{-1}$
A, B, D are orthogonal