CS111 F21 Homework 4

Michael Glushchenko, 9403890

October 24, 2021

I apologize for not formatting screenshots very well.

1 NCM 2.3: Member Forces

Here's the code I used, I believe the comments explain it well.

```
# equations are numbered from 0
# variables in the book are numbered from 1
alpha = 1/math.sqrt(2)
F = np.zeros((13,13)) # this will be our forces matrix
b = np.zeros(13) # this will be our member forces vector
b[[1,7,9]]=[10,15,20] # all zeros except equations 1, 7, and 9

# we proceed to make the equations as 2D arraus
eq0  = np.array([[1,5]    , [1,-1]]) # f2-f6=
eq1  = np.array([[2]      , [1]]) # f3=
eq2  = np.array([[0,3,4]  , [alpha,-1,-alpha]]) # alpha*f1-f4-alpha*f5=
eq3  = np.array([[0,2,4]  , [alpha,1,alpha]]) # alpha*f1+f3+alpha*f5=
eq4  = np.array([[3,7]    , [1,-1]]) # same logic for the rest
eq5  = np.array([[6]      , [1]])
eq6  = np.array([[4,5,8,9], [alpha,1,-alpha,-1]])
eq7  = np.array([[4,6,8]  , [alpha,1,alpha]])
eq8  = np.array([[9,12]   , [1,-1]])
eq9  = np.array([[10]     , [1]])
eq10 = np.array([[7,8,11] , [1,alpha,-alpha]])
eq11 = np.array([[8,10,11], [alpha,1,alpha]])
eq12 = np.array([[12,11]  , [1,alpha]])

# make an array of our equations so we can loop through them
eqArr = [eq0,eq1,eq2,eq3,eq4,eq5,eq6,eq7,eq8,eq9,eq10,eq11,eq12]
for i in range(F.shape[0]):
    F[i,eqArr[i][0][:].astype(int)] = eqArr[i][1][:]
#print(F) # to check the equation matrix looks correct
```

```
f_1 = npla.solve(F,b) # regular solve
f_2, relres = cs111.LUsolve(F,b) # check with cs111.LUsolve()
print("npla.solve(F,b) == cs111.LUsolve(F,b)?", (np.round(f_1,4)==np.round(f_2,4)).all()) # compare solutions
print("relative residual norm: ", relres, "\n")
print("Given the 'loads' vector b: \n", b)
print("Our vector of 'member forces' is \n", f_1)
```

```
npla.solve(F,b) == cs111.LUsolve(F,b)? True
relative residual norm:  2.0064672503295845e-16

Given the 'loads' vector b:
 [ 0. 10.  0.  0.  0.  0.  0. 15.  0. 20.  0.  0.  0.]
Our vector of 'member forces' is
 [-28.2843  20.      10.     -30.      14.1421  20.       0.     -30.
   7.0711  25.      20.     -35.3553  25.     ]
```

2 SPD Matrices

2.1

The first if statement below checks off the first 3 conditions. The if inside of the for loop checks off the last condition.

```
# first we find the A to satisfy conditions
x = np.zeros(2)
foundA = np.zeros((2,2))
found = False
while(not found):
    foundA = np.zeros((2,2))
    A = np.random.rand(2,2)
    Asymm = (A+A.T)/2
    n, m = Asymm.shape
    if(n == m and npla.matrix_rank(Asymm) == n and (Asymm[:,:] > 0).all()):
        print("Found A under given conditions: \n", Asymm)
        foundA = Asymm
        # then for this A we check 100 vectors to see if find one
        for i in range(100):
            temp = np.random.randn(2)
            if(temp.T@foundA@temp < 0):
                print("Found an x to satisfy x^T@A@x < 0: \n", temp)
                print("x^T@A@x = ", temp.T@foundA@temp)
                x = temp
                found = True
                break
        if(not found):
            print("Did not find such an x for this matrix A \n \n")
```

```
Found A under given conditions:
 [[0.8169 0.2842]
 [0.2842 0.2879]]
Did not find such an x for this matrix A


Found A under given conditions:
 [[0.6292 0.6543]
 [0.6543 0.2362]]
Found an x to satisfy x^T@A@x < 0:
 [ 0.142  -0.3759]
x^T@A@x =  -0.0238076816278393
```

2.2

$$A = B^T B,$$

We want to show $\forall x \neq 0, x^T Ax > 0$. We can rewrite

$$x^T Ax = x^T B^T Bx = (Bx)^T Bx = ||Bx||^2$$

We know the rank of matrix B is n, so its columns are linearly independent. We also know that $x \neq 0$. Thus,

$$||Bx||^2 > 0 \Rightarrow x^T Ax > 0 \Rightarrow \text{ A is SPD}$$

3 Cholesky Decomposition

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n} \\ \cdot & \cdot & & \\ \cdot & & \cdot & \\ 0 & 0 & \cdots & a_{n-1,n} \\ 0 & 0 & \cdots & a_{n,n} \end{pmatrix} = \begin{pmatrix} R_{1,1} & \cdots & 0 & 0 \\ R_{2,1} & \cdots & 0 & 0 \\ \cdot & \cdot & & \\ \cdot & & \cdot & \\ R_{n-1,1} & R_{n-1,2} & \cdots & 0 \\ R_{n,1} & R_{n,2} & \cdots & R_{n,n} \end{pmatrix} \begin{pmatrix} R_{1,1} & R_{2,1} & \cdots & R_{n,1} \\ 0 & R_{2,2} & \cdots & R_{n,2} \\ \cdot & \cdot & & \\ \cdot & & \cdot & \\ 0 & \cdots & & R_{n,n-1} \\ 0 & \cdots & 0 & R_{n,n} \end{pmatrix}$$

Start with an n-by-n matrix $A = R^T R$, multiply out $R^T R$, and set the resulting elements to the individual elements of the upper triangle of matrix A. The result involves the following equations, and the pattern goes on until the final row n:

$$\begin{array}{llll} r_{00} = \sqrt{a_{00}} & r_{11} = \sqrt{a_{11} - r_{01} r_{01}} & r_{22} = \sqrt{a_{22} - r_{02} r_{02} - r_{12} r_{12}} & \cdots \\ r_{01} = a_{01}/r_{00} & r_{12} = (a_{12} - r_{01} r_{02})/r_{11} & r_{23} = (a_{23} - r_{02} r_{03} - r_{12} r_{13})/r_{22} & \cdots \\ r_{02} = a_{02}/r_{00} & r_{13} = (a_{13} - r_{01} r_{03})/r_{11} & & \cdots \\ r_{03} = a_{03}/r_{00} & & & \cdots \end{array}$$

These warrant the following equations for diagonal and non-diagonal terms:

$$\forall i = j,\ L_{i,j} = \sqrt{a_{i,j} \ - \sum_{k=1}^{j-1} R_{i,k}^2}$$

$$\forall i \neq j,\ R_{i,j} = \frac{1}{R_{i,j}} (a_{i,j} - \sum_{k=1}^{j-1} R_{i,k} R_{j,k})$$

However, looking at the equations written out row by row, we can see a bit of logic here that is more than a head-on nested loop involving the calculation of every element of R. Here's the logic:

1. begin with R being a copy of the upper triangle of A
2. go through every row in R:
   - the top left-most term is just the square root of the current element in that cell
     - the rest of the row is divided by the scalar element we just calculated
   - we then update the rows below this one by subtracting out the portion of $\sum_{k=1}^{j-1} R_{i,k} R_{j,k}$ that we do have calculated

```
# Pre-condition: A is SPD
# Post-condition: Returns an upper triangular
# matrix R such that A=R.T@R
def Cfactor(A):
    n = A.shape[0]
    R = np.triu(A)
    for i in range(n):
        R[i, i] **= 0.5
        R[i, i+1:] /= R[i, i]
        for j in range(i + 1, n):
            R[j, j:] -= R[i, j] * R[i, j:]
    return R
```

Above is the algorithm we obtain by implementing the logic described earlier. Now, we check that $Cfactor()$ works correctly on a random 10-by-10 matrix:

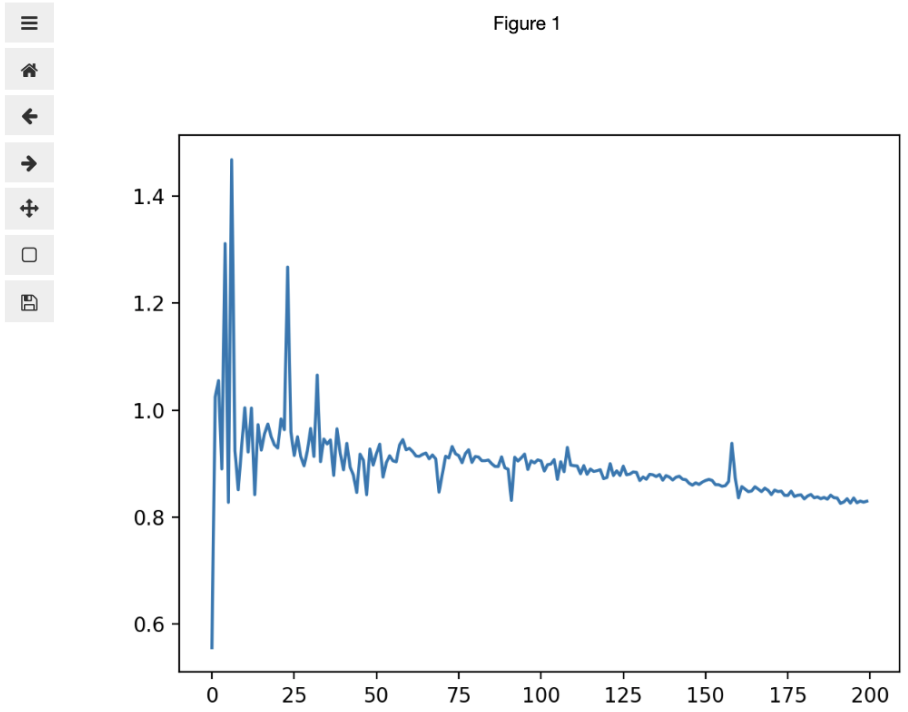
```
# Testing the Cholesky Decomposition Algorithm
# for correctness against cs111.LUsolve()
# test and inspect one random example
B = np.random.randn(10,10)
A = B.T @ B
b = np.random.randn(10)

LUsolveSolution, relres = cs111.LUsolve(A,b)

R = Cfactor(A)
y = cs111.Lsolve(R.T, b)
CholeskySolution = cs111.Usolve(R, y)

print("A = R.T@R? ", (np.round(R.T@R - A, 10) == 0).all(), "\n")
print("LUsolve Solution: \n", LUsolveSolution, "\n")
print("Cholesky Solution: \n", CholeskySolution, "\n")
print("Solutions equal? ", (np.round(LUsolveSolution-CholeskySolution, 6) == 0).all())

# check the correctness a couple more times
flag = True
for i in range(10000):
    B = np.random.randn(10,10)
    A = B.T @ B
    b = np.random.randn(10)
    LUsolveSolution, relres = cs111.LUsolve(A,b)
    R = Cfactor(A)
    CholeskySolution = cs111.Usolve(R, cs111.Lsolve(R.T, b))
    if (not (np.round(R.T@R - A, 6) == 0).all()):
        flag = False
print("Solution is correct" if flag else "Solution is wrong")
```

```
A = R.T@R?  True

LUsolve Solution:
 [ 0.2638 -1.8855 -0.7748 -1.3386 -0.2618 -0.1568  0.2993 -1.4988  0.4453
  0.5152]

Cholesky Solution:
 [ 0.2638 -1.8855 -0.7748 -1.3386 -0.2618 -0.1568  0.2993 -1.4988  0.4453
  0.5152]

Solutions equal?  True
Solution is correct
```

Finally, we time Cfactor() and cs111.LUfactorNoPiv for a range of 1000 values, every 5 values, and plot the ratio of Cfactor time over cs111.LUfactorNoPiv time:

```
# Timing Cfactor() against cs111.LUfactorNoPiv()
# plotting the ratio of the two runtimes against n
ratioResults = np.zeros(200)

for n in range(1, 1000, 5):
    B = np.random.randn(n,n)
    A = B.T @ B
    luTime = %timeit -n3 -r1 -q -p3 -o cs111.LUfactorNoPiv(A)
    cTime = %timeit -n3 -r1 -q -p3 -o Cfactor(A)
    ratioResults[int(n/5)] = cTime.average/luTime.average
```

```
# check some entries in the ratioResults,
ratioResults[150:]
# then plot it against size of matrix, n
plt.plot(range(200), ratioResults)
plt.show()
```



The result shows that Cfactor is faster than LUfactor. The ratio seems to get closer to 0.8-0.85, which is not quite 1/2, but it is more efficient.

4 Experimenting with Solvers

4.1 Which k in Under 30 Seconds?

I wrote the following function to find the value k for each solver that fits the description of the problem. Running k linearly in some range would warrant much too long of trials, thus we implement a sort of binary search type of approach.

```
def findThirtySecK(func, begin, k, end, cs111LUsolve = False):
    A = cs111.make_A_3D(k)
    if (cs111LUsolve):
        A = A.toarray()
    b = A @ np.ones(A.shape[0])
    t = %timeit -n1 -r1 -q -p3 -o func(A,b)
    if (end - begin <= 1 or (t.average > 28 and t.average < 30)):
        print("The largest value of k for which this solver")
        print("solves Ax=b within 30 seconds is: ", k, "\n")
        print("Solved in: ", t.average)
    elif (t.average < 28):
        begin = k
        k = round((begin + end) / 2)
        print("k = ", begin, " too quick, trying k = ", k)
        findThirtySecK(func, begin, k, end, cs111LUsolve)
    else:
        end = k
        k = round((begin + end) / 2)
        print("k = ", end, " too slow, trying k = ", k)
        findThirtySecK(func, begin, k, end, cs111LUsolve)
```

I chose a range of 28 and 30 because some solvers jump somewhat quickly in time, and I think capturing anything between the 2 values would suffice. As to the logic of the line that times the solver, we set the number of loops and the number of repetitions to 1, otherwise our code runs longer by a multiple of $r * n$, which in these cases would be more painful than useful (in terms providing the extra nanoseconds of precision by averaging out time trials).

```
arr = [cs111.CGsolve, cs111.Jsolve, scipy.sparse.linalg.cg, scipy.sparse.linalg.spsolve, cs111.LUsolve]
for item in arr:
    flag = False
    begin, k, end = 1, 100, 200
    funcName = item.__name__
    if (funcName == "spsolve"):
        begin, k, end = 1, 50, 100 # smaller range for the slower spsolve
    if (funcName == "LUsolve"):
        begin, k, end = 1, 15, 30 # an even smaller range for the slowest LUsolve
        flag = True
    print("Testing ", funcName, ":\n")
    findThirtySecK(item, begin, k, end, flag)
    print("\n\n")
```

```
Testing  CGsolve :

k =  100  too quick, trying k =  150
k =  150  too slow, trying k =  125
k =  125  too quick, trying k =  138
k =  138  too quick, trying k =  144
The largest value of k for which this solver
solves Ax=b within 30 seconds is:  144

Solved in:  29.340680711999994
```

The second snippet of code shown on the last page is the code that uses our function $findThirtySecondK()$ on each of the solvers; here's the rest of the outputs:

```
Testing  Jsolve :

k =  100  too quick, trying k =  150
k =  150  too slow, trying k =  125
k =  125  too slow, trying k =  112
k =  112  too slow, trying k =  106
The largest value of k for which this solver
solves Ax=b within 30 seconds is:  106

Solved in:  29.13135038900009

Testing  cg :

k =  100  too quick, trying k =  150
k =  150  too quick, trying k =  175
k =  175  too slow, trying k =  162
k =  162  too quick, trying k =  168
k =  168  too quick, trying k =  172
k =  172  too quick, trying k =  174
The largest value of k for which this solver
solves Ax=b within 30 seconds is:  174

Solved in:  28.71321223699988

Testing  spsolve :

k =  50  too slow, trying k =  26
k =  26  too quick, trying k =  38
k =  38  too quick, trying k =  44
k =  44  too slow, trying k =  41
k =  41  too quick, trying k =  42
k =  42  too slow, trying k =  42
The largest value of k for which this solver
solves Ax=b within 30 seconds is:  42

Solved in:  36.9148862269999

Testing  LUsolve :

k =  15  too slow, trying k =  8
k =  8  too quick, trying k =  12
k =  12  too quick, trying k =  14
k =  14  too quick, trying k =  14
The largest value of k for which this solver
solves Ax=b within 30 seconds is:  14

Solved in:  19.820786684999803
```

Here, already, the order of the speed of the 5 solvers can be seen fairly clearly by looking at how big of a matrix the solver can get through within the 30 second constraint. So far, the results seem to show that the fastest solver is $scipy.sparse.linalg.cg()$, while the dense $cs111.LUsolve()$ is (obviously) the slowest. $Jsolve()$ seems to be faster than LU factorization methods, but slower than the conjugate gradient method.

4.2 Accuracy & Speed Comparison with Constant k

For this section, a tolerance of $1e-10$ seems reasonable to use for consistency purposes. As for the k value, we choose 15 so that the slowest solver (cs111.LUsolve()) doesn't take too long to run. Here's the code used for this sub-question:

```
# testing speed and accuracy
# choose k = 15, tolerance = 10^-16, compare solvers
arr = [cs111.CGsolve, cs111.Jsolve, scipy.sparse.linalg.cg, scipy.sparse.linalg.spsolve, cs111.LUsolve]
names = arr
names = [item.__name__ for item in arr]
data = np.zeros((3,len(arr)))
k = 15

for i in range(len(arr)):
    name = arr[i].__name__
    A = cs111.make_A_3D(k)
    if (name == "LUsolve"):
        A = A.toarray()
    x_exact = np.ones(A.shape[0])
    b = A @ x_exact

    if (name == "CGsolve" or name == "Jsolve" or name == "cg"):
        t = %timeit -n1 -r1 -q -p3 -o arr[i](A,b,tol=1e-10)
        data[0, i] = t.average
        x = arr[i](A,b,tol=1e-10)[0]
        data[1, i] = npla.norm(b-A@x)/npla.norm(b)
        data[2, i] = npla.norm(x_exact-x)/npla.norm(x_exact)
    else:
        t = %timeit -n1 -r1 -q -p3 -o arr[i](A,b)
        data[0, i] = t.average
        if (name == "LUsolve"):
            x, relres = arr[i](A,b)
        else:
            x = arr[i](A,b)
            relres = npla.norm(b-A@x)/npla.norm(b)
        data[1, i] = relres
        data[2, i] = npla.norm(x_exact-x)/npla.norm(x_exact)
```

Putting all of this into a table, we get the following, where row 1 contains the times it took for the solver to run given our constraints, row 2 contains the the relative residual norm, and row 3 contains the relative error norm:

```
print(tabulate(data, headers=names, tablefmt="grid"))
```

```
+-------------+-------------+-------------+-------------+-------------+
|     CGsolve |      Jsolve |          cg |     spsolve |      LUsolve |
+=============+=============+=============+=============+=============+
| 0.00545795  | 0.0765003   | 0.00299479  | 0.0416165   | 33.5182      |
+-------------+-------------+-------------+-------------+-------------+
| 3.92664e-11 | 4.835e-10   | 3.92664e-11 | 2.48315e-15 |  6.79081e-15 |
+-------------+-------------+-------------+-------------+-------------+
| 6.86191e-12 | 2.98519e-09 | 6.8619e-12  | 4.38191e-16 |  2.17806e-15 |
+-------------+-------------+-------------+-------------+-------------+
```

LUsolve() is very obviously the slowest solver, the *scipy cg* algorithm seems, once again, to be the fastest. Overall, results line up with **4.1** so far.

4.3 Accuracy & Speed Comparison with Varying k

Now, we set our tolerance equal to, once again, $1e-10$. This time, we run the solvers for multiple values of k, so that we can see how the table produced in **4.2** changes as the value of k changes. First, we run loop of k values in the range 1 to 20, choosing every 4th k to, once again, save some time here.

```
arr = [cs111.CGsolve, cs111.Jsolve, scipy.sparse.linalg.cg, scipy.sparse.linalg.spsolve, cs111.LUsolve]
names = arr
names = [item.__name__ for item in arr]
data = np.zeros((3,len(arr)))
for k in range(1,20,4):
    for i in range(len(arr)):
        name = arr[i].__name__
        A = cs111.make_A_3D(k)
        if (name == "LUsolve"):
            A = A.toarray()
        x_exact = np.ones(A.shape[0])
        b = A @ x_exact

        if (name == "CGsolve" or name == "Jsolve" or name == "cg"):
            t = %timeit -n1 -r1 -q -p3 -o arr[i](A,b,tol=1e-10)
            data[0, i] = t.average
            x = arr[i](A,b,tol=1e-10)[0]
            data[1, i] = npla.norm(b-A@x)/npla.norm(b)
            data[2, i] = npla.norm(x_exact-x)/npla.norm(x_exact)
        else:
            t = %timeit -n1 -r1 -q -p3 -o arr[i](A,b)
            data[0, i] = t.average
            if (name == "LUsolve"):
                x, relres = arr[i](A,b)
            else:
                x = arr[i](A,b)
                relres = npla.norm(b-A@x)/npla.norm(b)
            data[1, i] = relres
            data[2, i] = npla.norm(x_exact-x)/npla.norm(x_exact)
    print("k = ", k, "\n", tabulate(data, headers=names, tablefmt="grid"), "\n\n")
```

Here is the output this produces for the second and last values that run in this range, $k = 5$ and $k = 17$:

```
k =  5
+-------------+-------------+-------------+-------------+-------------+
|     CGsolve |      Jsolve |          cg |     spsolve |     LUsolve |
+=============+=============+=============+=============+=============+
| 0.000384    | 0.00555654  | 0.000632041 | 0.000409092 | 0.0369408   |
+-------------+-------------+-------------+-------------+-------------+
| 8.02103e-16 | 8.79907e-11 | 4.54956e-16 | 8.56704e-16 | 1.01969e-15 |
+-------------+-------------+-------------+-------------+-------------+
| 2.73034e-16 | 1.60873e-10 | 2.13209e-16 | 2.53364e-16 | 3.78519e-16 |
+-------------+-------------+-------------+-------------+-------------+

k =  17
+-------------+-------------+-------------+-------------+--------------+
|     CGsolve |      Jsolve |          cg |     spsolve |      LUsolve |
+=============+=============+=============+=============+==============+
| 0.0124621   | 0.103124    | 0.00565118  | 0.0924774   | 92.5706      |
+-------------+-------------+-------------+-------------+--------------+
| 5.83203e-11 | 2.44683e-08 | 5.83204e-11 | 2.95379e-15 |  8.44494e-15 |
+-------------+-------------+-------------+-------------+--------------+
| 1.37769e-11 | 1.77242e-07 | 1.37769e-11 | 8.0327e-16  |  3.4644e-15  |
+-------------+-------------+-------------+-------------+--------------+
```

$LUsolve()$ getting slower with k growing, by a relatively little amount. The residual and error norms of the LU factorization solvers were not affected; the precision of the rest of the solvers worsened with a growing k.

Since *cs*111.*LUsolve*() was getting slow so quick, we can take it out of the array *arr* of functions that we want to test; we can then expand the testing range for k to 50ish, the value around which *spsolve*() begins to run quite a bit slower:

```
# Now we remove LUsolve from the array of functions we want to test
# this way we can increase k more and really compare speed and precision
# of all the other solvers
arr = [cs111.CGsolve, cs111.Jsolve, scipy.sparse.linalg.cg, scipy.sparse.linalg.spsolve]
names = arr
names = [item.__name__ for item in arr]
data = np.zeros((3,len(arr)))
for k in range(1,42,10):
    for i in range(len(arr)):
        name = arr[i].__name__
        A = cs111.make_A_3D(k)
        if (name == "LUsolve"):
            A = A.toarray()
        x_exact = np.ones(A.shape[0])
        b = A @ x_exact

        if (name == "CGsolve" or name == "Jsolve" or name == "cg"):
            t = %timeit -n1 -r1 -q -p3 -o arr[i](A,b,tol=1e-10)
            data[0, i] = t.average
            x = arr[i](A,b,tol=1e-10)[0]
            data[1, i] = npla.norm(b-A@x)/npla.norm(b)
            data[2, i] = npla.norm(x_exact-x)/npla.norm(x_exact)
        else:
            t = %timeit -n1 -r1 -q -p3 -o arr[i](A,b)
            data[0, i] = t.average
            if (name == "LUsolve"):
                x, relres = arr[i](A,b)
            else:
                x = arr[i](A,b)
                relres = npla.norm(b-A@x)/npla.norm(b)
            data[1, i] = relres
            data[2, i] = npla.norm(x_exact-x)/npla.norm(x_exact)
    print("k = ", k, "\n", tabulate(data, headers=names, tablefmt="grid"), "\n\n")
```

Here are the outputs for the second and last k in this range:

```
k =  11
 +-------------+-------------+-------------+-------------+
|     CGsolve |      Jsolve |          cg |     spsolve |
+=============+=============+=============+=============+
| 0.00409857  | 0.0311927   | 0.00200459  | 0.00989809  |
+-------------+-------------+-------------+-------------+
| 8.29406e-11 | 9.94388e-11 | 8.29406e-11 | 1.85748e-15 |
+-------------+-------------+-------------+-------------+
| 1.13702e-11 | 4.19477e-10 | 1.13702e-11 | 5.35393e-16 |
+-------------+-------------+-------------+-------------+

k =  41
 +-------------+------------+-------------+--------------+
|     CGsolve |     Jsolve |          cg |      spsolve |
+=============+============+=============+==============+
| 0.190944    | 1.24577    | 0.0945781   | 28.7824      |
+-------------+------------+-------------+--------------+
| 9.87564e-11 | 0.00192319 | 9.87564e-11 |  8.34064e-15 |
+-------------+------------+-------------+--------------+
| 3.33125e-11 | 0.0459409  | 3.33125e-11 |  4.03499e-15 |
+-------------+------------+-------------+--------------+
```

We can now notice that as k grows more, *spsolve*() is the next function to become the slowest at solving such matrices; however, it's residual and error norm

remain unaffected by the size of k. The same can't be said about $Jsolve()$, which very clearly has much worse precision here, although its computing time remains fast.
We now remove $spsolve()$ from arr, and run the code again, on a new range of values k, from 1 to 100, every 20th k. Here's the output for the second and last k's from this range:

```
k =  21
 +-------------+-------------+-------------+
|     CGsolve |      Jsolve |          cg |
+=============+=============+=============+
| 0.0169864   | 0.185146    | 0.00875327  |
+-------------+-------------+-------------+
| 7.93236e-11 | 2.93861e-06 | 7.93235e-11 |
+-------------+-------------+-------------+
| 1.51347e-11 | 2.80628e-05 | 1.51347e-11 |
+-------------+-------------+-------------+

k =  81
 +-------------+-------------+-------------+
|     CGsolve |      Jsolve |          cg |
+=============+=============+=============+
| 3.35418     | 13.7541     | 1.70513     |
+-------------+-------------+-------------+
| 9.20227e-11 |  0.00587873 | 9.20226e-11 |
+-------------+-------------+-------------+
| 7.35965e-11 |  0.357805   | 7.35965e-11 |
+-------------+-------------+-------------+
```

$Jsolve()$ is the next algorithm to start slowing down, as k edges to 100. The precision of this solver goes down substantially, especially visible in the relative error norm. The conjugate gradient solvers still only take a few seconds each, and their precision remains high.

Finally, we take out $Jsolve()$ from arr, run the code again, and display the second and last k tested in the range, the range for this final step being 1 to 150 every 25 k's:

```
k =  50
 +-------------+-------------+
|     CGsolve |          cg |
+=============+=============+
| 0.42884     | 0.23298     |
+-------------+-------------+
| 9.02992e-11 | 9.02991e-11 |
+-------------+-------------+
| 4.40113e-11 | 4.40113e-11 |
+-------------+-------------+
```

```
k =  125
 +-------------+-------------+
|      CGsolve |           cg |
+==============+==============+
| 22.2413      | 11.1334      |
+--------------+--------------+
|  9.09323e-11 |  9.09323e-11 |
+--------------+--------------+
|  7.6562e-11  |  7.65619e-11 |
+--------------+--------------+
```

It's noticeable that the $cs111$ conjugate gradient solver is now as fast as the *scipy* conjugate gradient solver; in fact, they seem to differ by a factor of 2. Both of these solvers, once again, have high precision, low relative residual norms, and low relative error norms. This concludes the close look at the behavior of the 5 given solvers.