CS111 F21 Homework 3

Michael Glushchenko, 9403890

October 16, 2021

1 Solving $A^T x = b$

When solving $Ax = b$, we used $(PA = LU) \Rightarrow (A = P^T LU)$ to substitute:

$$Ax = b \Rightarrow (P^T LU)x = b \Rightarrow LUx = Pb$$

Then we set $y = Ux$ and solved $Ly = Pb$ by using

$$y = Lsolve(L, b[p], unit\_diag = True)$$

After, we got our solution by using

$$x = Usolve(U, y)$$

Now we're solving $A^T x = b$. Transpose both sides of the $LU = PA$ equation:

$$(LU = PA) \Rightarrow ((LU)^T = (PA)^T)$$

Simplify

$$\Rightarrow U^T L^T = A^T P^T \Rightarrow A^T = U^T L^T P$$

Substitute

$$A^T x = b \Rightarrow (U^T L^T P)x = b$$

Following previous logic, we set $y = L^T Px$ and use

$$y = Lsolve(U^T, b)$$

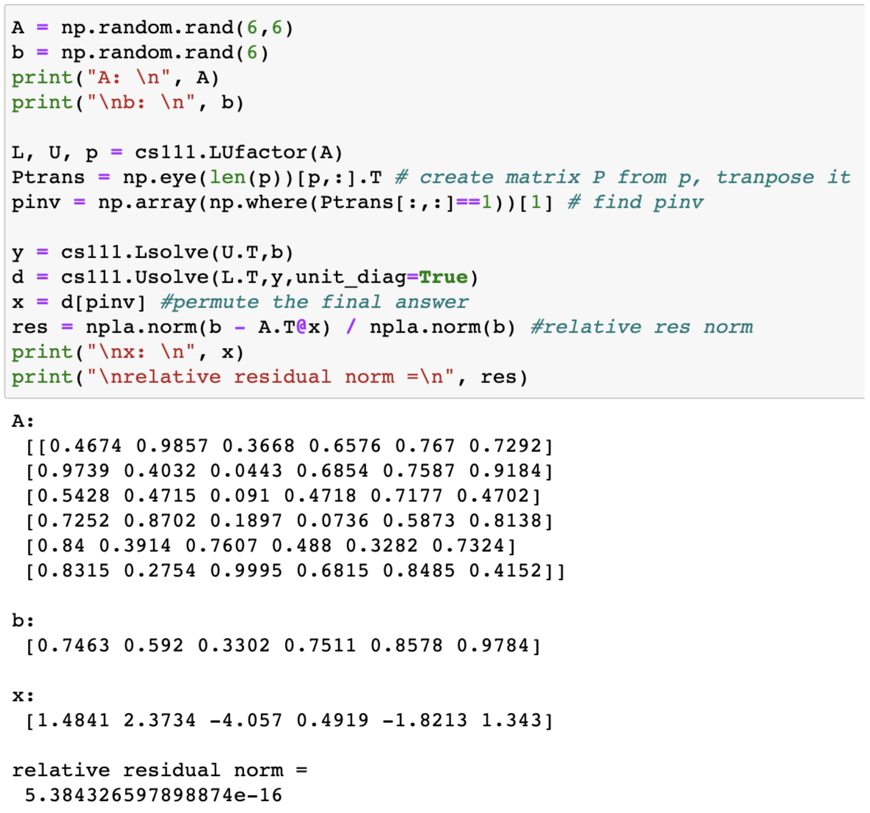
Then we set $d = Px$ and solve $L^T d = y$ via

$$d = Usolve(L^T, y, unit\_diag = True)$$

Vector d should simply be a permutation of the sought-after solution x:

$$d = Px \Rightarrow x = P^T d$$

In Python, we generate our random matrix A and random vector b. After calling $cs111.LUfactor(A)$ we now have L, U, and p. Following the logic described above, we check that our answer is correct. Here's the code and output:



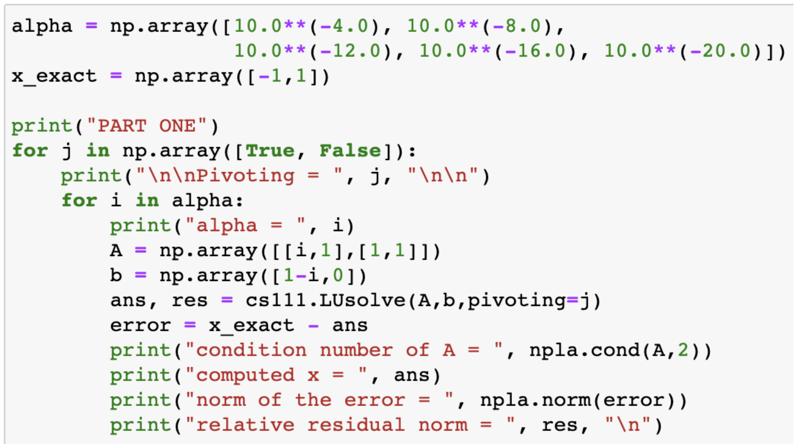
```python
A = np.random.rand(6,6)
b = np.random.rand(6)
print("A: \n", A)
print("\nb: \n", b)

L, U, p = cs111.LUfactor(A)
Ptrans = np.eye(len(p))[p,:].T # create matrix P from p, tranpose it
pinv = np.array(np.where(Ptrans[:,:]==1))[1] # find pinv

y = cs111.Lsolve(U.T,b)
d = cs111.Usolve(L.T,y,unit_diag=True)
x = d[pinv] #permute the final answer
res = npla.norm(b - A.T@x) / npla.norm(b) #relative res norm
print("\nx: \n", x)
print("\nrelative residual norm =\n", res)
```

```
A:
 [[0.4674 0.9857 0.3668 0.6576 0.767  0.7292]
 [0.9739 0.4032 0.0443 0.6854 0.7587 0.9184]
 [0.5428 0.4715 0.091  0.4718 0.7177 0.4702]
 [0.7252 0.8702 0.1897 0.0736 0.5873 0.8138]
 [0.84   0.3914 0.7607 0.488  0.3282 0.7324]
 [0.8315 0.2754 0.9995 0.6815 0.8485 0.4152]]

b:
 [0.7463 0.592  0.3302 0.7511 0.8578 0.9784]

x:
 [1.4841 2.3734 -4.057  0.4919 -1.8213 1.343]

relative residual norm =
 5.384326597898874e-16
```

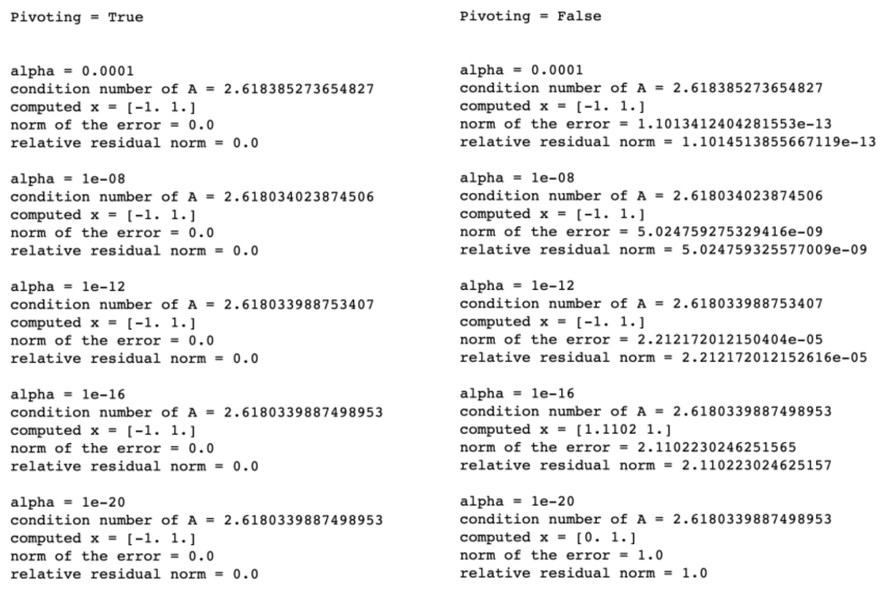
2 Condition Number of A

2.1

For this part, we have: $\begin{pmatrix} \alpha & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1-\alpha \\ 0 \end{pmatrix}$ for some $\alpha < 1$.
The code and output for this part are as follows:



```
alpha = np.array([10.0**(-4.0), 10.0**(-8.0),
                  10.0**(-12.0), 10.0**(-16.0), 10.0**(-20.0)])
x_exact = np.array([-1,1])

print("PART ONE")
for j in np.array([True, False]):
    print("\n\nPivoting = ", j, "\n\n")
    for i in alpha:
        print("alpha = ", i)
        A = np.array([[i,1],[1,1]])
        b = np.array([1-i,0])
        ans, res = cs111.LUsolve(A,b,pivoting=j)
        error = x_exact - ans
        print("condition number of A = ", npla.cond(A,2))
        print("computed x = ", ans)
        print("norm of the error = ", npla.norm(error))
        print("relative residual norm = ", res, "\n")
```



```
Pivoting = True


alpha = 0.0001
condition number of A = 2.618385273654827
computed x = [-1. 1.]
norm of the error = 0.0
relative residual norm = 0.0

alpha = 1e-08
condition number of A = 2.618034023874506
computed x = [-1. 1.]
norm of the error = 0.0
relative residual norm = 0.0

alpha = 1e-12
condition number of A = 2.618033988753407
computed x = [-1. 1.]
norm of the error = 0.0
relative residual norm = 0.0

alpha = 1e-16
condition number of A = 2.6180339887498953
computed x = [-1. 1.]
norm of the error = 0.0
relative residual norm = 0.0

alpha = 1e-20
condition number of A = 2.6180339887498953
computed x = [-1. 1.]
norm of the error = 0.0
relative residual norm = 0.0
```

```
Pivoting = False


alpha = 0.0001
condition number of A = 2.618385273654827
computed x = [-1. 1.]
norm of the error = 1.1013412404281553e-13
relative residual norm = 1.1014513855667119e-13

alpha = 1e-08
condition number of A = 2.618034023874506
computed x = [-1. 1.]
norm of the error = 5.024759275329416e-09
relative residual norm = 5.024759325577009e-09

alpha = 1e-12
condition number of A = 2.618033988753407
computed x = [-1. 1.]
norm of the error = 2.212172012150404e-05
relative residual norm = 2.212172012152616e-05

alpha = 1e-16
condition number of A = 2.6180339887498953
computed x = [1.1102 1.]
norm of the error = 2.1102230246251565
relative residual norm = 2.110223024625157

alpha = 1e-20
condition number of A = 2.6180339887498953
computed x = [0. 1.]
norm of the error = 1.0
relative residual norm = 1.0
```

Clearly, the matrix stays well-conditioned regardless of α or truth value of *pivoting*. When $pivoting = True$, the relative residual norm is 0 for all α that we tested, as our answer seems to be exact. When $pivoting = False$, however, the relative residual norm grows as α decreases (without *pivoting* the pivot used is very small, thus being too close to 0, thus making us close to division by 0 during $LU - factorization$, thus furthering us from the exact answer).

2.2

For this part, we have: $\begin{pmatrix} 1+\alpha & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} -\alpha \\ 0 \end{pmatrix}$ for some $\alpha < 1$.
The code and output for this part are as follows:

```
alpha = np.array([10.0**(-4.0), 10.0**(-8.0),
                  10.0**(-12.0), 10.0**(-16.0), 10.0**(-20.0)])
x_exact = np.array([-1,1])

print("PART TWO")
for j in np.array([True, False]):
    print("\n\nPivoting = ", j, "\n\n")
    for i in alpha:
        print("alpha = ", i)
        A = np.array([[1+i,1],[1,1]])
        b = np.array([-i,0])
        print("condition number of A = ", npla.cond(A,2))
        try:
            ans, res = cs111.LUsolve(A,b,pivoting=j)
            error = x_exact - ans
            print("computed x = ", ans)
            print("norm of the error = ", npla.norm(error))
            print("relative residual norm = ", res, "\n")
        except:
            print("Answer not returned \n")
```

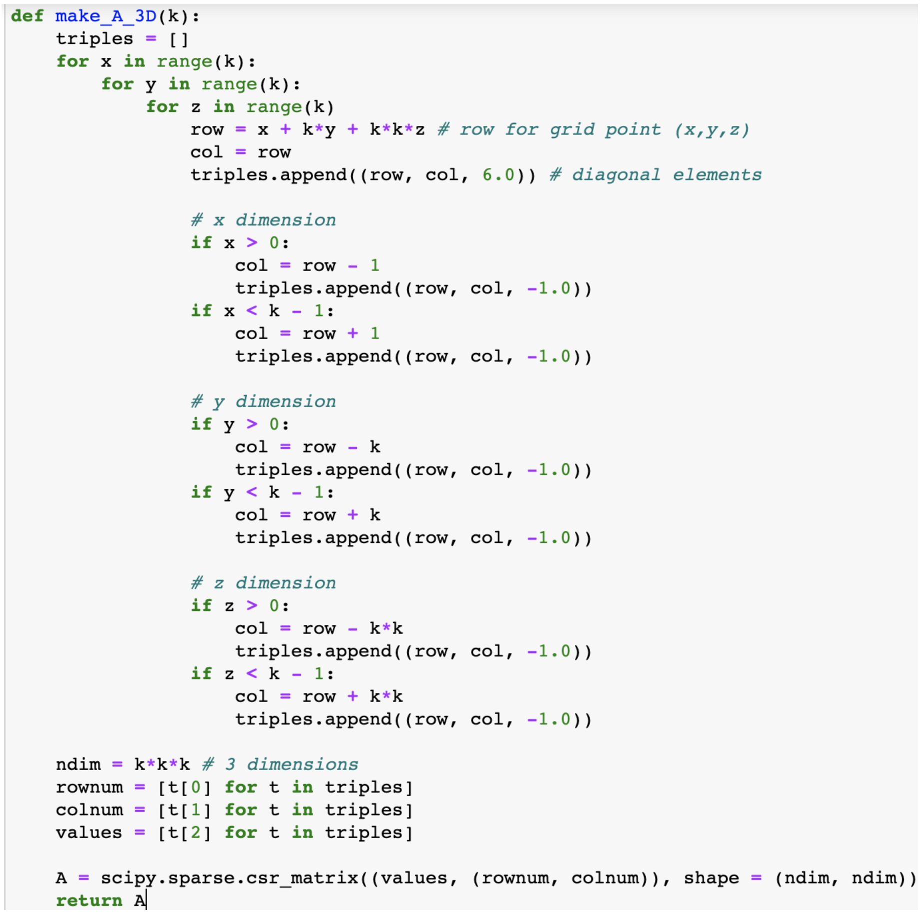
```
Pivoting = True


alpha = 0.0001
condition number of A = 40002.00007504555
computed x = [-1. 1.]
norm of the error = 2.3488583181909273e-13
relative residual norm = 1.1018204577883939e-13

alpha = 1e-08
condition number of A = 399999992.40613085
computed x = [-1. 1.]
norm of the error = 1.0153631235080597e-08
relative residual norm = 6.0774709918447105e-09

alpha = 1e-12
condition number of A = 3997936426769.096
computed x = [-0.9999 0.9999]
norm of the error = 0.00012571323468403843
relative residual norm = 2.212172012148393e-05

alpha = 1e-16
condition number of A = 5.961777047638983e + 16
Answer not returned

alpha = 1e-20
condition number of A = 5.961777047638983e + 16
Answer not returned
```

```
alpha = 0.0001
condition number of A = 40002.00007504555
computed x = [-1. 1.]
norm of the error = 2.3488583181909273e-13
relative residual norm = 1.1018204577883939e-13

alpha = 1e-08
condition number of A = 399999992.40613085
computed x = [-1. 1.]
norm of the error = 1.0153631235080597e-08
relative residual norm = 6.0774709918447105e-09

alpha = 1e-12
condition number of A = 3997936426769.096
computed x = [-0.9999 0.9999]
norm of the error = 0.00012571323468403843
relative residual norm = 2.212172012148393e-05

alpha = 1e-16
condition number of A = 5.961777047638983e + 16
Answer not returned

alpha = 1e-20
condition number of A = 5.961777047638983e + 16
Answer not returned
```

In the 3 tries where solutions were returned, we can see the relative residual norm increasing (same reason as **2.1**). As α decreases, matrix A becomes *ill-conditioned* under both $pivoting$ and $non-pivoting$ solutions; the matrix gets closer to having $row_1 = row_2$, making one of the $pivots$ very small. In trials 4 and 5, we finally get too close to dividing by 0 during $LU-factorization$. Running $cs111.LUfactor(A)$ results in a non-zero pivot not being found, rendering the factorization impossible, and making $cs111.LUsolve(A, b)$ crash.

3 3D Temperature Matrix

Here's my code for $make\_A\_3D(k)$:



```
def make_A_3D(k):
    triples = []
    for x in range(k):
        for y in range(k):
            for z in range(k)
                row = x + k*y + k*k*z # row for grid point (x,y,z)
                col = row
                triples.append((row, col, 6.0)) # diagonal elements

                # x dimension
                if x > 0:
                    col = row - 1
                    triples.append((row, col, -1.0))
                if x < k - 1:
                    col = row + 1
                    triples.append((row, col, -1.0))

                # y dimension
                if y > 0:
                    col = row - k
                    triples.append((row, col, -1.0))
                if y < k - 1:
                    col = row + k
                    triples.append((row, col, -1.0))

                # z dimension
                if z > 0:
                    col = row - k*k
                    triples.append((row, col, -1.0))
                if z < k - 1:
                    col = row + k*k
                    triples.append((row, col, -1.0))

    ndim = k*k*k # 3 dimensions
    rownum = [t[0] for t in triples]
    colnum = [t[1] for t in triples]
    values = [t[2] for t in triples]

    A = scipy.sparse.csr_matrix((values, (rownum, colnum)), shape = (ndim, ndim))
    return A
```

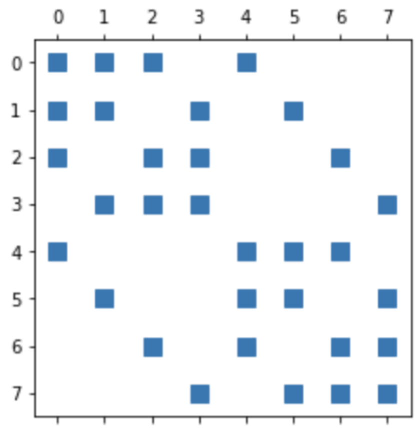
Starting on the next page, we go through each k in the range $[2, 3, 4, 5]$ by using the following code:

```
for k in range(2,6):
    A = make_A_3D(k)
    print("k:", k)
    print("dimensions:", A.shape)
    print("nonzeros:", A.nnz)
    #print("A as sparse matrix:\n", A)
    print("A as dense matrix:\n", A.toarray())
    plt.spy(A)
```

3.1 $k = 2$

Here, we get the following dense matrix A and chart:

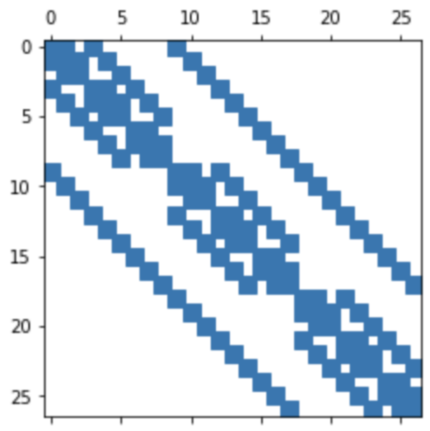
```
k: 2
dimensions: (8, 8)
nonzeros: 32
A as dense matrix:
 [[6. -1. -1. 0. -1. 0. 0. 0.]
 [-1. 6. 0. -1. 0. -1. 0. 0.]
 [-1. 0. 6. -1. 0. 0. -1. 0.]
 [0. -1. -1. 6. 0. 0. 0. -1.]
 [-1. 0. 0. 0. 6. -1. -1. 0.]
 [0. -1. 0. 0. -1. 6. 0. -1.]
 [0. 0. -1. 0. -1. 0. 6. -1.]
 [0. 0. 0. -1. 0. -1. -1. 6.]]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



3.2 $k = 3$

Here, we look at the sparse matrix A, since the dense matrix becomes too large:

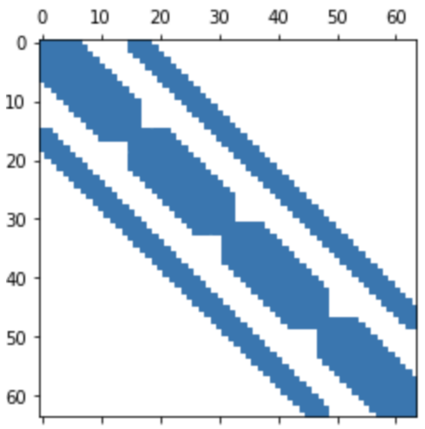
```
k: 3
dimensions: (27, 27)
nonzeros: 135
A as sparse matrix:        ::
    (0, 0) 6.0             (21, 24) -1.0
  (0, 1) -1.0              (22, 13) -1.0
  (0, 3) -1.0              (22, 19) -1.0
  (0, 9) -1.0              (22, 21) -1.0
  (1, 0) -1.0              (22, 22) 6.0
  (1, 1) 6.0               (22, 23) -1.0
  (1, 2) -1.0              (22, 25) -1.0
  (1, 4) -1.0              (23, 14) -1.0
  (1, 10) -1.0             (23, 20) -1.0
  (2, 1) -1.0              (23, 22) -1.0
  (2, 2) 6.0               (23, 23) 6.0
  (2, 5) -1.0              (23, 26) -1.0
  (2, 11) -1.0             (24, 15) -1.0
  (3, 0) -1.0              (24, 21) -1.0
  (3, 3) 6.0               (24, 24) 6.0
  (3, 4) -1.0              (24, 25) -1.0
  (3, 6) -1.0              (25, 16) -1.0
  (3, 12) -1.0             (25, 22) -1.0
  (4, 1) -1.0              (25, 24) -1.0
  (4, 3) -1.0              (25, 25) 6.0
  (4, 4) 6.0               (25, 26) -1.0
  (4, 5) -1.0              (26, 17) -1.0
  (4, 7) -1.0              (26, 23) -1.0
  (4, 13) -1.0             (26, 25) -1.0
  (5, 2) -1.0              (26, 26) 6.0
```

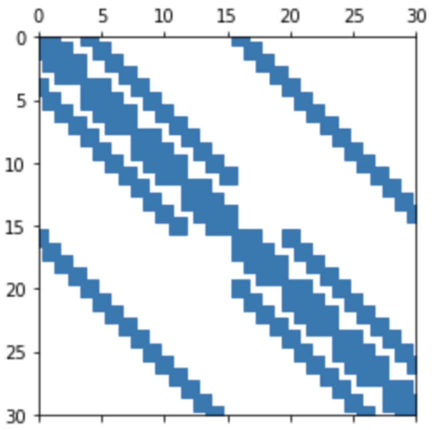


3.3 $k = 4$

```
x_min = 0
x_max = 30
y_min = 30
y_max = 0
plt.axis([x_min, x_max, y_min, y_max])
```

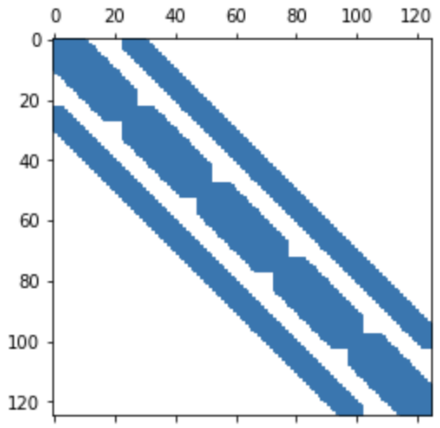
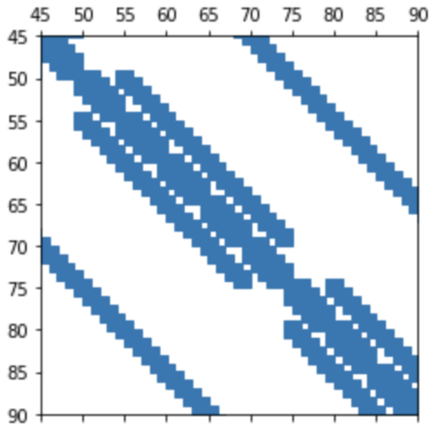
Zooming in produces the following plots (zoomed chart on the right):





3.4 $k = 5$

Now, we adjust the coordinates and inspect $k = 5$:


0
20
40
60
80
100
120

45
50
55
60
65
70
75
80
85
90