Homework set 2

Please **submit this Jupyter notebook through Canvas** no later than **Mon Nov. 13, 9:00**. **Submit the notebook file with your answers (as .ipynb file) and a pdf printout. The pdf version can be used by the teachers to provide feedback. A pdf version can be made using the save and export option in the Jupyter Lab file menu.**

Homework is in **groups of two**, and you are expected to hand in original work. Work that is copied from another group will not be accepted.

Exercise 0

Write down the names + student ID of the people in your group.

Koen Weverink (14711982) & Jasper Timmer (12854328)

Importing packages

Execute the following statement to import the packages `numpy`, `math` and `scipy.sparse`. If additional packages are needed, import them yourself.

In [ ]:
```
import math
import numpy as np
import scipy.sparse as sp
from scipy import linalg
from time import time
import sys
```

Sparse matrices

A matrix is called sparse if only a small fraction of the entries is nonzero. For such matrices, special data formats exist. `scipy.sparse` is the scipy package that implements such data formats and provides functionality such as the LU decomposition (in the subpackage `scipy.sparse.linalg`).

As an example, we create the matrix

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

in the so called compressed sparse row (CSR) format. As you can see, the arrays `row`, `col`, `data` contain the row and column coordinate and the value of each nonzero element respectively.

In [ ]:
```
# a sparse matrix with 6 nonzero entries
row = np.array([0, 0, 1, 2, 2, 3])
col = np.array([0, 2, 1, 2, 3, 3])
data = np.array([1.0, 2, 3, 4, 5, 6])
sparseA = sp.csr_array((data, (row, col)), shape=(4, 4))

# convert to a dense matrix. This allows us to print to screen in regular formatting
denseA = sparseA.toarray()
print(denseA)
```

```
[[1. 0. 2. 0.]
 [0. 3. 0. 0.]
 [0. 0. 4. 5.]
 [0. 0. 0. 6.]]
```

For sparse matrices, a sparse data format is much more efficient in terms of storage than the standard array format. Because of this efficient storage, very large matrices of size $n \times n$ with $n = 10^7$ or more can be stored in RAM for performing computations on regular computers. Often the number of nonzero elements per row is quite small, such as 10's or 100's nonzero elements per row. In a regular, dense format, such matrices would require a supercomputer or could not be stored.

In the second exercise you have to use the package `scipy.sparse`, please look up the functions you need (or ask during class).

Heath computer exercise 2.1

(a)

Show that the matrix

$$A = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix}.$$

is singular. Describe the set of solutions to the system $Ax = b$ if

$$b = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.5 \end{bmatrix}.$$

(N.B. this is a pen-and-paper question.)

A is singular if $\det(A) = 0$.

$$\det(A) =$$

$$\begin{vmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{vmatrix} =$$

$$0.1 \cdot \begin{vmatrix} 0.5 & 0.6 \\ 0.8 & 0.9 \end{vmatrix} - 0.4 \cdot \begin{vmatrix} 0.2 & 0.3 \\ 0.8 & 0.9 \end{vmatrix} + 0.7 \cdot \begin{vmatrix} 0.1 & 0.3 \\ 0.5 & 0.6 \end{vmatrix} =$$

$$0.1 \cdot (0.5 \cdot 0.9 - 0.8 \cdot 0.6) - 0.4 \cdot (0.2 \cdot 0.9 - 0.8 \cdot 0.3) + 0.7 \cdot (0.2 \cdot 0.6 - 0.5 \cdot 0.3) =$$

$$0.1 \cdot (0.45 - 0.48) - 0.4 \cdot (0.18 - 0.24) + 0.7 \cdot (0.12 - 0.15) \ =$$

$$0.1 \cdot -0.03 - 0.4 \cdot -0.06 + 0.7 \cdot -0.03 =$$

$$-0.003 + 0.024 - 0.021 =$$

$$0$$

Hence, A is singular, which means that $Ax = b$ does either have no solution at all, or infinitely many solutions.

(b)

If we were to use Gaussian elimination with partial pivoting to solve this system using exact arithmetic, at what point would the process fail?

To see which is the case, we try to solve it as usual:

Swapping the first row with the third row

$$\begin{aligned} 0.7x_1 + 0.8x_2 + 0.9x_3 &= 0.5 \\ 0.4x_1 + 0.5x_2 + 0.6x_3 &= 0.3 \\ 0.1x_1 + 0.2x_2 + 0.3x_3 &= 0.1 \end{aligned}$$

Normalizing first row

$$\begin{aligned} x_1 + \frac{8}{7}x_2 + \frac{9}{7}x_3 &= \frac{5}{7} \\ 0.4x_1 + 0.5x_2 + 0.6x_3 &= 0.3 \\ 0.1x_1 + 0.2x_2 + 0.3x_3 &= 0.1 \end{aligned}$$

Substracting the first row 0.4 times from the second row and 0.1 times from the third row:

$$\begin{aligned} x_1 + \frac{8}{7}x_2 + \frac{9}{7}x_3 &= \frac{5}{7} \\ \frac{3}{70}x_2 - \frac{3}{35}x_3 &= 0.3 - \frac{2}{7} \\ \frac{3}{35}x_2 - \frac{6}{35}x_3 &= 0.1 - \frac{1}{14} \end{aligned}$$

Dividing the third row by 2 will make it identical to the second row. This means there are infinitely many solutions to this problem.

(c)

Because some of the entries of A are not exactly representable in a binary floating point system, the matrix is no longer exactly singular when entered into a computer; thus, solving the system by Gaussian elimination will not necessarily fail. Solve this system on a computer using a library routine for Gaussian elimination. Compare the computed solution with your description of the solution set in part (a). What is the estimated value for $\text{cond}(A)$? How many digits of accuracy in the solution would this lead you to expect?

In [ ]:
```
A = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]])

pl, u = linalg.lu(A, permute_l=True)

print(u)

cond = np.linalg.cond(A)
print(f"\nThe condition number of A is {cond}, which should lead to around {np.log10(cond)} digits of accuracy.")
```

```
[[7.00000000e-01 8.00000000e-01 9.00000000e-01]
 [0.00000000e+00 8.57142857e-02 1.71428571e-01]
 [0.00000000e+00 0.00000000e+00 1.11022302e-16]]

The condition number of A is 2.37588029981422e+16, which should lead to around 16.37582455649447 digits of accuracy.
```

Heath computer exercise 2.17

Consider a horizontal cantilevered beam that is clamped at one end but free along the remainder of its length. A discrete model of the forces on the beam yields a system of linear equations $Ax = b$, where the $n \times n$ matrix A has the banded form

$$\begin{bmatrix} 9 & -4 & 1 & 0 & \cdots & \cdots & 0 \\ -4 & 6 & -4 & 1 & \ddots & & \vdots \\ 1 & -4 & 6 & -4 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -4 & 6 & -4 & 1 \\ \vdots & & \ddots & 1 & -4 & 5 & -2 \\ 0 & \cdots & \cdots & 0 & 1 & -2 & 1 \end{bmatrix},$$

the n-vector b is the known load on the bar (including its own weight), and the n-vector x represents the resulting deflection of the bar that is to be determined. We will take the bar to be uniformly loaded, with $b_i = 1/n^4$ for each component of the load vector.

(a)

Make a python function that creates the matrix A given the size n.

In [ ]:
```python
import numpy as np

def create_beam_matrix(n):
    # Initialize the matrix with zeros
    A = np.zeros((n, n))

    # Fill the diagonal and off-diagonal elements
    for i in range(n):
        A[0, 0] = 9
        A[i, i] = 6

        if i > 0:
            A[i, i-1] = -4
            A[i-1, i] = -4
        if i > 1:
            A[i, i-2] = 1
            A[i-2, i] = 1
```

```
    A[-2, -3:] = [-4, 5, -2]
    A[-1, -3:] = [1, -2, 1]
    return A

n = 5
A_matrix = create_beam_matrix(n)
print(A_matrix)
```

```
[[ 9. -4.  1.  0.  0.]
 [-4.  6. -4.  1.  0.]
 [ 1. -4.  6. -4.  1.]
 [ 0.  1. -4.  5. -2.]
 [ 0.  0.  1. -2.  1.]]
```

(b)

Solve this linear system using both a standard library routine for dense linear systems and a library routine designed for sparse linear systems. Take $n = 100$ and $n = 1000$. How do the two routines compare in the time required to compute the solution? And in the memory occupied by the LU decomposition? (Hint: as part of this assignment, look for the number of nonzero elements in the matrices L and U of the sparse LU decomposition.)

In [ ]:
```
for n in [100, 1000]:
    # Get Parameters
    b = []
    for n_i in range(n):
        b_vals = 1/((n_i + 1)**4)
        b.append(b_vals)

    # Create the beam matrix with nxn rows and columns
    A = create_beam_matrix(n)

    start = time()
    # Get the sparse A
    sparseA = sp.csr_array(A)
    sparse_solve = sp.linalg.spsolve(sparseA, b)
    end = time() - start
    print("n={} Sparse solve took {} seconds".format(n, end))
```

```
    # convert to a dense matrix
    denseA = sparseA.toarray()
    start = time()
    dense_solve = linalg.solve(denseA, b)
    end = time() - start
    print("n={} Dense solve took {} seconds".format(n, end))

    lu = sp.linalg.splu(sparseA)
    memory_occupied_u = sys.getsizeof(lu.U)
    memory_occupied_l = sys.getsizeof(lu.L)
    print("n={} Sparse memory occupation of LU decomp. Upper: {} Lower: {}".format(n, memory_occupied_u, memory_occupied_l))

    _, l, u = linalg.lu(denseA)
    memory_occupied_u = sys.getsizeof(u)
    memory_occupied_l = sys.getsizeof(l)
    print("n={} Dense memory occupation of LU decomp. Upper: {} Lower: {}\n".format(n, memory_occupied_u, memory_occupied_l))
```

```
n=100 Sparse solve took 0.0 seconds
n=100 Dense solve took 0.09380674362182617 seconds
n=100 Sparse memory occupation of LU decomp. Upper: 48 Lower: 48
n=100 Dense memory occupation of LU decomp. Upper: 80128 Lower: 80128
```

```
c:\Users\koenw\AppData\Local\Programs\Python\Python310\lib\site-packages\scipy\sparse\linalg\_dsolve\linsolve.py:394: SparseE
fficiencyWarning: splu converted its input to CSC format
  warn('splu converted its input to CSC format', SparseEfficiencyWarning)
n=1000 Sparse solve took 0.016366243362426758 seconds
n=1000 Dense solve took 0.1998438835144043 seconds
n=1000 Sparse memory occupation of LU decomp. Upper: 48 Lower: 48
n=1000 Dense memory occupation of LU decomp. Upper: 8000128 Lower: 8000128
```

As can be seen, the solving of the sparse matrix is significantly faster than the solving of the dense matrix. This is true for both the 100x100 matrix and the 1000x1000 matrix. Furthermore, the memory occupation for the LU decomposition is significantly lower for the sparse matrix. Moreover, the memory occupation for the sparse matrix does, in this case, not change whether its a 100x100 matrix or 1000x1000, whereas there is a very big change in memory occupation in the dense matrix.

(c)

For $n = 100$, what is the condition number? What accuracy do you expect based on the condition number?

In [ ]:
```
n = 100
A = create_beam_matrix(n)
cond = np.linalg.cond(A)
print(f"\nThe condition number of A is {cond}, which could lead to a loss of around {np.log10(cond)} digits of accuracy.")
```

```
The condition number of A is 130661079.61972737, which could lead to a loss of around 8.11614624233499 digits of accuracy.
```

We expect a very low accuracy based on our calculated condition number. If the condition number is very large, as it is in our case, it implies that a small change in the input data can lead a large change in the solution. This means that the problem is 'ill-conditioned', and the accuracy of the solution is bad.

(d)

How well do the answers of (b) agree with each other (make an appropriate quantitative comparison)?

Should we be worried about the fact that the two answers are different?

In [ ]:
```
# Compute cosine similarity between vectors a and b
def compute_cosine_sim(a, b):
    return np.dot(a, b)/(np.linalg.norm(a) * np.linalg.norm(b))

similarity = compute_cosine_sim(dense_solve, sparse_solve)
print(similarity)
```

```
0.9999999999999857
```

A cosine similarity of 0.9999999999999857 is extremely close to 1, which indicates a very high similarity between the two arrays being compared. Therefore, we should not be worried about the fact that the two anwers are different. Moreover, seeing that calculations using sparse matrices is both faster and requires less computer memory, it will likely be better to us sparce metrices if there are many 0-values.