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.

Zijian Zhang, 14851598

Lina Xiang, 14764369

Importing packages

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

In [1]:
```
import math
import numpy as np
import scipy.sparse as sp
```

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 [2]:
```
# 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 fo
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.)

Answer:

The rank of A can be obtained by performing row transformations on A:

$$A = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix} \rightarrow \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0 & -0.3 & -0.6 \\ 0 & -0.6 & -1.2 \end{bmatrix} \rightarrow \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0 & -0.3 & -0.6 \\ 0 & 0 & 0 \end{bmatrix}.$$

Therefore, $\text{rank}(A) = 2 < 3$, which means A is singular.

The solution set of the system $Ax = b$ can be obtained by the following steps:

$$A|b = \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0.4 & 0.5 & 0.6 & 0.3 \\ 0.7 & 0.8 & 0.9 & 0.5 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0 & -0.3 & -0.6 & -0.1 \\ 0 & -0.6 & -1.2 & -0.2 \end{array}\right] \rightarrow \left[\begin{array}{cc} 0.1 & 0.2 \\ 0 & -0.3 \\ 0 & 0 \end{array}\right]$$

$$0.1x_1 + 0.2x_2 + 0.3x_3 = 0.1, \quad -0.3x_2 - 0.6x_3 = -0.1.$$

Therefore, there are ∞ set of solutions in solving phase. Set $x_1 = \lambda$, then the solutions can be represented as

$$x = \begin{bmatrix} 0 \\ 1 \\ -\frac{1}{3} \end{bmatrix} + \lambda \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}.$$

(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?

Answer:

The LU factorization still works, and we can obtain the upper triangular system

$$M_2P_2M_1P_1Ax = \begin{bmatrix} 0.7 & 0.8 & 0.9 \\ 0 & 3/35 & 6/35 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1/35 \\ 0 \end{bmatrix} = M_2P_2M_1P_1b.$$

However, the process would fail in back-substitution due to the 0 on diagonal.

(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 $\mathrm{cond}(A)$? How many digits of accuracy in the solution would this lead you to expect?

In [3]:
```
from numpy.linalg import solve, cond, norm, inv
import sys

A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6],
              [0.7, 0.8, 0.9]])
b = np.array([0.1, 0.3, 0.5]).T
x = solve(A, b)
x1, x2, x3 = x
cond_A = norm(A) * norm(inv(A))
```

```
print(f"Computed solution: x1={x1}, x2={x2}, x3={x3}")
print(f"Our description: x1={x1}, x2={1-2*x1}, x3={x1-1/3}")
print(f"Relative error: x2_rel_error={abs(x2-(1-2*x1))/(1-2*x1)}, x3_rel_er
print(f"cond(A)={cond(A)}, cond_A={cond_A}")
print("epsilon = {}".format(sys.float_info.epsilon))
```

```
Computed solution: x1=0.06327985739750433, x2=0.8734402852049913, x3=-0.270
053475935829
Our description: x1=0.06327985739750433, x2=0.8734402852049914, x3=-0.27005
347593582896
Relative error: x2_rel_error=1.2710920751320664e-16, x3_rel_error=-2.055561
441632715e-16
cond(A)=2.37588029981422e+16, cond_A=6.830458856948811e+16
epsilon = 2.220446049250313e-16
```

Answer:

When λ takes the value of x_1 in computed solution, the computed solution are almost the same as our description of the solution set in part (a), with only a relative error of the order of 10^{-16} due to computational accuracy.

The estimated value for cond(A) is $2.37588029981422 \times 10^{16}$, and the machine precision $\epsilon_{\text{mach}} = 2.220446049250313 \times 10^{-16}$, which means $\text{cond}(A)\epsilon_{\text{mach}} \approx 0$. Therefore, we would expect no digits of accuracy in the solution in double precision arithmetic.

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

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

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 [4]:
```python
def generate_dense_A(n):
    A = np.zeros((n, n))

    # Main diagonal with 6, except the first element and the last two elemer
    A[np.arange(1, n, 2), np.arange(1, n, 2)] = 1
    A[np.arange(n), np.arange(n)] = 6
    A[0, 0] = 9
    A[-2, -2] = 5
    A[-1, -1] = 1

    # Upper and lower diagonals with 4, except the last elements
    A[np.arange(n - 1), np.arange(1, n)] = -4
    A[np.arange(1, n), np.arange(n - 1)] = -4
    A[n - 2, n - 1] = -4  # Last element of the upper diagonal
    A[n - 1, n - 2] = -4  # Last element of the lower diagonal
    A[-1, -2] = -2
    A[-2, -1] = -2

    A[np.arange(n - 2), np.arange(2, n)] = 1
    A[np.arange(2, n), np.arange(n - 2)] = 1

    return A

A = generate_dense_A(4)
print(A)
```

```
[[ 9. -4.  1.  0.]
 [-4.  6. -4.  1.]
 [ 1. -4.  5. -2.]
 [ 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 [5]:
```python
def compare_dense_sparse(n, ifprint = True):
    A_dense = generate_dense_A(n)
    b = np.full(n, 1 / n**4)

    # Convert A to a sparse matrix
    A_sparse = sp.csc_matrix(A_dense)

    if ifprint:
        print(f"n={n}")
        # Dense matrix
        print("time_dense:")
        %timeit solve(A_dense, b)
        m_dense = 3 * n**2

        # Sparse matrix
        print("time_sparse:")
        %timeit sp.linalg.spsolve(A_sparse, b)
        LU = sp.linalg.splu(A_sparse)
        m_sparse = LU.L.nnz + LU.U.nnz
```

```
        # Compare the memory occupancy
        print("memory_dense={:.1e}, memory_sparse={}".format(m_dense, m_spar

    return solve(A_dense, b), sp.linalg.spsolve(A_sparse, b)


x_d_100, x_s_100 = compare_dense_sparse(100)
print()
x_d_1000, x_s_1000 = compare_dense_sparse(1000)
```

```
n=100
time_dense:
428 µs ± 80.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
time_sparse:
49.8 µs ± 431 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
memory_dense=3.0e+04, memory_sparse=691

n=1000
time_dense:
19.8 ms ± 2.72 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
time_sparse:
331 µs ± 14.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
memory_dense=3.0e+06, memory_sparse=6990
```

Answer:

We use `numpy.linalg.solve` (hereafter referred to as `solve`) for solving dense linear system, and `scipy.sparse.linalg.spsolve` (hereafter referred to as `spsolve`) for solving sparse linear system. It is found that under the same n, `spsolve` is much faster than `solve`. Moreover, as n increases, the time required by `solve` grows super-linearly, whereas the increase for spsolve is sub-linear.

In the context of estimating memory occupation during LU decomposition, distinct methods are applied to `solve` and `spsolve`. In the case of `solve`, where matrix A is dense, the decomposition process involves storing three $n \times n$ matrices in memory. Therefore, we use the total number of entries in the dense matrices A, L, and U, which amounts to $3n^2$, as an indicator of the memory used by `solve`. Conversely, for `spsolve`, where A is a sparse matrix, only the non-zero entries are stored. Given that the number of non-zero entries in A is considerably fewer than those in L and U, we consider the total number of non-zero elements in L and U as an indicator of the memory used by `spsolve`.

As the results of the code run show, `solve` takes up far less memory than `spsolve`. Moreover, as n increases by a factor of 10, the memory occupied by `solve` grows by a factor of $10^2 = 100$, while the memory occupied by `solve` grows linearly.

(c)

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

In [6]:
```
A = generate_dense_A(100)
cond_number = np.linalg.cond(A)
epsilon = sys.float_info.epsilon
print("Condition number of the matrix: {:.2e}".format(cond_number))
```

```
print("epsilon_mach = {:.2e}".format(epsilon))
print("accuracy = cond * epsilon_mach = {:.2e}".format(epsilon * cond_numbe
```

```
Condition number of the matrix: 1.31e+08
epsilon_mach = 2.22e-16
accuracy = cond * epsilon_mach = 2.90e-08
```

Answer:

The condition number is about 1.31×10^8. The estimated accuracy is about 2.9×10^{-8} , which means we would expect 7 digits of accuracy in the solution.

(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 [7]:
```
x_d_100, x_s_100 = compare_dense_sparse(100, ifprint=False)
x_d_1000, x_s_1000 = compare_dense_sparse(1000, ifprint=False)

A = generate_dense_A(100)
cond_number = np.linalg.cond(A)
accuracy_100 = cond_number * epsilon

A = generate_dense_A(1000)
cond_number = np.linalg.cond(A)
accuracy_1000 = cond_number * epsilon

diff_100_1 = norm(x_d_100 - x_s_100, ord=1) / norm(x_s_100, ord=1)
diff_1000_1 = norm(x_d_1000 - x_s_1000, ord=1) / norm(x_s_1000, ord=1)

inf = float('inf')
diff_100_inf = norm(x_d_100 - x_s_100, ord=inf) / norm(x_s_100, ord=inf)
diff_1000_inf = norm(x_d_1000 - x_s_1000, ord=inf) / norm(x_s_1000, ord=inf)

print("accuracy_100={:.2e}, diff_100_1={:.2e}, diff_100_inf={:.2e}".format(
    accuracy_100, diff_100_1, diff_100_inf))
print("accuracy_1000={:.2e}, diff_1000_1={:.2e}, diff_1000_inf={:.2e}".forma
    accuracy_1000, diff_1000_1, diff_1000_inf))
```

```
accuracy_100=2.90e-08, diff_100_1=3.52e-11, diff_100_inf=2.91e-11
accuracy_1000=2.88e-04, diff_1000_1=2.19e-07, diff_1000_inf=2.23e-07
```

Answer:

The relative difference derived from both the 1-norm and infinity norm are significantly lower than the anticipated level of accuracy. Herein, the 1-norm accumulates all the differences, whereas the infinity norm focuses solely on the largest discrepancy. Consequently, these discrepancies are negligible and can be disregarded in the context of this analysis.