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.

Pablo Alves - 15310191 **Nitai Nijholt** - 12709018

Importing packages

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

In [139…

```
import math
import numpy as np
import scipy.sparse as sp
from scipy.linalg import lu_factor, lu_solve
import sys
import pandas as pd
```

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 [140…

```
# 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 regul
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.i) Showing A is singular

It will suffice to show that $det(A) = 0$.

By simple inspection we see that $R_3 = 2 \cdot R_2 - R_1$.

Because the third row of A is a linear combination of the previous two rows, this in turn implies $det(A) = 0$

Which in turns determines that A is singular.

■

(a.ii) Describing the set of solutions

Observing the $Ax = b$ system we want to solve, we notice that we can first simplify it.

Taking the common term $1/10$ out of both A and b and cancelling it, we are then left with:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix}$$

Giving rise to equations:

$x + 2y + 3z = 1$

$4x + 5y + 6z = 3$

$7x + 8y + 9z = 5.$

Solving for x in first equation yields $x = 1 - 2y - 3z$

Substituting this in the second equation yields $3y + 6z = 1$

Which yields $y = -(6z - 1)/3$

Substituting y back in the x equation yields $x = 1 - 3z + 4z - 1/3$

Which yields $x = -9z + 5/3$

Therefore our infinite solutions will be of the form:

$x = -9z + 5/3$

$y = -(6z - 1)/3$

$z = z$

■

(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?

(b) Answer

Let

$$A' = (A|b) = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 3 \\ 7 & 8 & 9 & 5 \end{pmatrix}$$

We first try to create a zero at element $a'_{21} = 4$

For this we compute $R_2 \leftarrow R_3/2 + R_1/2 = (R_3 + R_1)/2$

Yielding:

$$A' = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 0 & 0 & 0 & 0 \\ 7 & 8 & 9 & 5 \end{pmatrix}$$

We then try to create a zero at element $a'_{31} = 7$

For this we compute $R_3 \leftarrow R_3 - 7 \cdot R_1$

Yielding:

$$A' = \begin{pmatrix} 1 & 2 & 3 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & -6 & -12 & -2 \end{pmatrix}$$

Our next element is $a'_{32} = -6$

However, the process fails here, as it is not possible to create a zero in this position.

■

(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?

(c.i) Solve the system with a library routine for Gaussian elimination

In [141…

```
# Define A and b
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])

# Compute LU decomposition
lu, piv = lu_factor(A)

# Solve the system
x = lu_solve((lu, piv), b)

# Estimate cond(A)
cond_number = np.linalg.cond(A)

# Print results
print('A:',  A)
print('b',   b)
print('LU',  lu)
print('piv', piv)
print('x',   x)
print('Condition number:', cond_number)
print(np.log10(cond_number))
```

```
A: [[0.1 0.2 0.3]
 [0.4 0.5 0.6]
 [0.7 0.8 0.9]]
b [0.1 0.3 0.5]
LU [[7.00000000e-01 8.00000000e-01 9.00000000e-01]
 [1.42857143e-01 8.57142857e-02 1.71428571e-01]
 [5.71428571e-01 5.00000000e-01 1.11022302e-16]]
piv [2 2 2]
x [ 0.16145833  0.67708333 -0.171875  ]
Condition number: 2.1118968335779856e+16
16.324672699040686
```

(c.ii) Compare the computed solution with your description of the solution set in part (a).

Unlike the solution obtained in part (a), the compution performed yields a unique solution for x, which is mathematically inaccurate.

(c.iii) What is the estimated value for $\mathrm{cond}(A)$?

The estimated value is $2.1118968335779856 \cdot 10^{16}$

(c.iv) How many digits of accuracy in the solution would this lead you to expect?

In our case, because the exponent in our condition number is 16, we expect to loose at least about $\log_{10}(cond(A))$ digits of accuracy in our result [1] , which in this case is 16 digits.

Because the solution values in x are small (within -1 and 1), this renders our result basically useless in terms of accuracy.

Intuitively, our condition number reflects the fact that the output x values of the system vary greatly to a small change in the input matrix A,

which is an unexpected behavior for a simple system like this one which is computed assuming a unique solution for x.

Thus, this huge condition number indicates that our system does not actually have a unique solution.

In short, this example illustrates the importance of analyzing the results of our computations and how the condition number can be used as an indicator in systems of linear equations.

[1] For a detailed explanation of the underlying math, see Heath, M. T. (2018). Scientific computing: an introductory survey, revised second edition. Society for Industrial and Applied Mathematics. p.60

(c.v) EXTRA: Solving the system after simplifying it first

We will illustrate this point further by repeating the previous computation on the equivalent system that results from first simplifying the $1/10$ term, as done in part (a)

In [142…

```
print('Results when simplifying A and b first:')

# Define A and b
A = 10*np.array([[0.1,0.2,0.3],[0.4,0.5,0.6],[0.7,0.8,0.9]])
```

```
b = 10*np.array([0.1,0.3,0.5])

# Compute LU decomposition
lu, piv = lu_factor(A)

# Solve the system
x = lu_solve((lu, piv), b)

# Estimate cond(A)
cond_number = np.linalg.cond(A)

# Print results
print('A:',  A)
print('b',   b)
print('LU',  lu)
print('piv', piv)
print('x',   x)
print('Condition number:', cond_number)
```

```
Results when simplifying A and b first:
A: [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
b [1. 3. 5.]
LU [[7.         8.         9.        ]
 [0.14285714 0.85714286 1.71428571]
 [0.57142857 0.5        0.        ]]
piv [2 2 2]
x [ nan -inf  inf]
Condition number: 3.813147060626918e+16
```

```
C:\Users\nitai\AppData\Local\Temp\ipykernel_32432\132216981.py:8: LinAlg
Warning: Diagonal number 3 is exactly zero. Singular matrix.
  lu, piv = lu_factor(A)
```

In this case, and unlike the previous computation, the library used now gives a warning when printing the solution,

indicating that our original matrix A was singular, which is consistent with our previous results.

In particular, when solving the system, the *diagonal number 3 [of the matrix] is exactly zero.*,

because now the elements in A and b of our equivalent system are not loosing accuracy due to approximations arising due to their storing in the computer.

This additional computation we perfomed highlights that:

1. Innacuracies of the storing method of decimal numbers can give rise to innacuracies in the results,

2. Mathematically equivalent systems can give rise to different computations,

3. Using a mathematically equivalent system can simplify result interpretation of limit cases

4. Understanding the interplay between the mathematical model and its computation is important to properly evaluate the accuracy of its results

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 [143…

```python
def make_matrix(n) -> np.ndarray:
    """Returns an n x n matrix of banded form representing a beam clamp

    # create a matrix of zeroes
    matrix_with_zeroes = np.zeros((n,n))

    # define the constant elements
    constant_elements = [1,-4,6,-4,1]

    # create a copy of the matrix with zeroes to be filled in
    matrix_filled = matrix_with_zeroes
```

```
    # loop over the matrix and fill in the constant elements in and aro
    for i in range(n-4):
        matrix_filled[i+2,i:i+5] = constant_elements

    # fill in beginning elements of the matrix
    matrix_filled[0,0:3] = 9, -4, 1
    matrix_filled[1,0:4] = -4, 6, -4, 1

    # fill in ending elements of the matrix
    matrix_filled[n-2,n-4:] = 1, -4, 5, -2
    matrix_filled[n-1,n-3:] = 1, -2, 1

    return matrix_filled
```

In [144…

```
# testing the function
test_matrix = make_matrix(8)
test_matrix
```

Out[144…

```
array([[ 9., -4.,  1.,  0.,  0.,  0.,  0.,  0.],
       [-4.,  6., -4.,  1.,  0.,  0.,  0.,  0.],
       [ 1., -4.,  6., -4.,  1.,  0.,  0.,  0.],
       [ 0.,  1., -4.,  6., -4.,  1.,  0.,  0.],
       [ 0.,  0.,  1., -4.,  6., -4.,  1.,  0.],
       [ 0.,  0.,  0.,  1., -4.,  6., -4.,  1.],
       [ 0.,  0.,  0.,  0.,  1., -4.,  5., -2.],
       [ 0.,  0.,  0.,  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 [145…

```
def make_b(n) -> np.ndarray:
    """ Returns a vector b consisting of n components. Each component b
    b = np.ones(n)/n**4
    return b
```

In [146…

```
# instantiate the matrices
A_n_100 = make_matrix(100)
A_n_1000 = make_matrix(1000)
```

In [147…

```
# instantiate the b vectors for n = 100 and n = 1000
b_100 = make_b(100)
b_1000 = make_b(1000)
```

In [148…
```
# LU factorize the matrix using standard scipy library routines for den
L_100, U_100 = lu_factor(A_n_100)
```

In [149…
```
# LU factorize the matrix using standard scipy library routines for den
L_1000, U_1000 = lu_factor(A_n_1000)
```

In [150…
```
# LU factorize the matrices using standard scipy library routines for s
LU_100_sparse = sp.linalg.splu(sp.csc_matrix(A_n_100))
```

In [151…
```
# LU factorize the matrices using standard scipy library routines for s
LU_1000_sparse = sp.linalg.splu(sp.csc_matrix(A_n_1000))
```

In [152…
```
# Using SciPy library routines for DENSE matrices solving the systems o
x_100_dense = lu_solve((L_100, U_100), b_100)
```

In [153…
```
# Using SciPy library routines for SPARSE matrices solving the systems
x_1000_dense = lu_solve((L_1000, U_1000), b_1000)
```

In [154…
```
# Using SciPy library routines for SPARSE matrices solving the systems
x_100_sparse = LU_100_sparse.solve(b_100)
```

In [155…
```
# Using SciPy library routines for SPARSE matrices solving the systems
x_1000_sparse = LU_1000_sparse.solve(b_1000)
```

Cell 11 (dense method) has an execution time of 1.6 seconds in case of n = 1000 vs. cell 13 (sparse method) has an execution time of 0.0 seconds for n = 1000. For n = 100, the dense method has an execution time of 0.4 (see cell 10) and sparse methods have a time of 0.0 (see cell 12). This seems to indicate that LU factorization is faster using sparse methods and this difference increases as n increases given this matrix, values of n and experimental setup. Solving time does not differ in timing significantly, taking 0.0 seconds for both sparse and dense methods and for both n = 100 and n = 1000, as can be seen from cells 14, 15, 16 and 17. Next, the running time is measured in more detail.

In [156…
```
# creating a function to time the LU factorization
def time_lu_decomposition(A, mode='dense'):
    """Returns the time it takes to factorize a matrix A using LU decom
    if mode == 'dense':
        L, U = lu_factor(A)
        return L, U
    elif mode == 'sparse':
        LU = sp.linalg.splu(sp.csc_matrix(A))
        return LU
    else:
        raise ValueError('mode must be either dense or sparse')
```

Next line magic is used to time the cell executions over 3 runs, 10 loops each

In [157…
```
%%capture time_lu_decomposition_dense_100
%timeit -r 3 -n 10 time_lu_decomposition(A_n_100, 'dense')
```

In [158…
```
%%capture time_lu_decomposition_dense_1000
%timeit -r 3 -n 10 time_lu_decomposition(A_n_1000, 'dense')
```

In [159…
```
%%capture time_lu_decomposition_sparse_100
%timeit -r 3 -n 10 time_lu_decomposition(A_n_100, 'sparse')
```

In [160…
```
%%capture time_lu_decomposition_sparse_1000
%timeit -r 3 -n 10 time_lu_decomposition(A_n_1000, 'sparse')
```

In [161…
```
# Show the results in a dataframe
pd.set_option('max_colwidth', 400)
df = pd.DataFrame({'n': [100, 1000], 'dense': [time_lu_decomposition_de
df.set_index('n', inplace=True)
df
```

Out[161…

n	dense	sparse
100	11.7 ms +- 4.64 ms per loop (mean +- std. dev. of 3 runs, 10 loops each	246 us +- 112 us per loop (mean +- std. dev. of 3 runs, 10 loops each
1000	95.7 ms +- 37.8 ms per loop (mean +- std. dev. of 3 runs, 10 loops each	4.7 ms +- 49.6 us per loop (mean +- std. dev. of 3 runs, 10 loops each

These results show that, indeed, the sparse method is faster for LU factorization given this matrix, as confirmed in both chosen settings of n, with the difference becoming larger as n increases.

In [162…
```
def count_non_0_elements(matrix: np.ndarray) -> int:
    """Returns the number of non-zero elements in a matrix"""
    return np.count_nonzero(matrix)
```

In [163…
```
# the .nnz method returns non-zero elements in a sparse matrix, see htt
print('non zero elements of L + U, n = 100, sparse:', LU_100_sparse.nnz
print('non zero elements of L + U, n = 1000, sparse:', LU_1000_sparse.n
print('non zero elements of L + U, n = 100, dense:', count_non_0_elemen
print('non zero elements of L + U, n = 1000, dense:', count_non_0_eleme
```

```
non zero elements of L + U, n = 100, sparse: 784
non zero elements of L + U, n = 1000, sparse: 7083
non zero elements of L + U, n = 100, dense: 9902
non zero elements of L + U, n = 1000, dense: 999002
```

In [164…
```
# print the space taken up in memory by the LU factorization of the spa
memory_space_taken_LU_sparse_100 = sys.getsizeof(LU_100_sparse)
memory_space_taken_LU_sparse_1000 = sys.getsizeof(LU_1000_sparse)
memory_space_taken_LU_dense_100 = sys.getsizeof(L_100 + U_100)
```

```
memory_space_taken_LU_dense_1000 = sys.getsizeof(L_1000 + U_1000)
print(f'memory_space_taken_LU_sparse_100: {memory_space_taken_LU_sparse
print(f'memory_space_taken_LU_sparse_1000: {memory_space_taken_LU_spars
print(f'memory_space_taken_LU_dense_100: {memory_space_taken_LU_dense_1
print(f'memory_space_taken_LU_dense_1000: {memory_space_taken_LU_dense_
```

```
memory_space_taken_LU_sparse_100: 144 bytes
memory_space_taken_LU_sparse_1000: 144 bytes
memory_space_taken_LU_dense_100: 80128 bytes
memory_space_taken_LU_dense_1000: 8000128 bytes
```

The number of non zero elements is significantly higher using dense methods, resulting in larger object size in the memory.

(c)

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

In [165…

```
# Condition number of A_n_100
cond_100 = np.linalg.cond(A_n_100)
print('condition:', cond_100)
print('Digits accuracy lost :', np.log10(cond_100))
```

```
condition: 130661079.38449307
Digits accuracy lost : 8.116146241553112
```

Condition number is 130661079.38449307 so we expect to lose $\log_{10}(cond(A))$, 8 digits of accuracy. Relating this to the result of 1c.

(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 [166…

```
def compare_solutions(x_1, x_2) -> np.ndarray:
    """Returns the difference between two vectors x_1 and x_2"""
    return np.linalg.norm(x_1 - x_2)
```

In [167…

```
# comparing the eudclidian

difference_solutions_100 = compare_solutions(x_100_dense, x_100_sparse)
difference_solutions_1000 = compare_solutions(x_1000_dense, x_1000_spar
print(f'difference_solutions_100: {difference_solutions_100}')
print(f'difference_solutions_1000: {difference_solutions_1000}')
```

```
difference_solutions_100: 2.943350696865687e-10
difference_solutions_1000: 3.406242239657442e-07
```

In this case the difference in solutions across methods still appears quite small although it seems to increase by 3 orders of magnitude when n is increased to 1000. Next we compare the quality of solutions by computing the residual, r:

$$residual = b - A\hat{x}$$

In [169…
```
# comparing the accuracy of solutions by computing the residual
residual_100_sparse = np.linalg.norm(b_100 - A_n_100 @ x_100_sparse)
residual_1000_sparse = np.linalg.norm(b_1000 - A_n_1000 @ x_1000_sparse
residual_100_dense = np.linalg.norm(b_100 - A_n_100 @ x_100_dense)
residual_1000_dense = np.linalg.norm(b_1000 - A_n_1000 @ x_1000_dense)
print(f'residual_100_sparse: {residual_100_sparse}')
print(f'residual_100_dense: {residual_100_dense}')
print(f'residual_1000_sparse: {residual_1000_sparse}')
print(f'residual_1000_dense: {residual_1000_dense}')

# differences in resisduals
# take the factor difference maybe instead of the difference in residua
print(f'ratio of residuals for n = 100: {(residual_100_dense/residual_1
print(f'ratio of residuals for n = 1000: {residual_1000_dense/residual_
```

```
residual_100_sparse: 5.422800026827077e-16
residual_100_dense: 6.213745308942541e-16
residual_1000_sparse: 1.5786692371158656e-15
residual_1000_dense: 1.6321437794182021e-15
ratio of residuals for n = 100: 1.145855513425276
ratio of residuals for n = 1000: 1.0338731768790475
```

In [172…
```
machine_epsilon = np.finfo(float).eps
print('machine_precision:', machine_epsilon)
```

```
machine_precision: 2.220446049250313e-16
```

The fact that we use a computer for this problem in the first place implies that we are content with an error at the machine precision level which the residuals are very close to, so NO, we should not be worried.

More generally however, if we should be worried about this difference in solutions depends on the accuracy we require of the answer in order to achieve the goal for which the calculation was done. From the residuals, we can see that the difference in the answer is negligable as the residuals are small (5.422×10^{-16} for sparse & n = 100, vs. 6.214×10^{-16} for dense & n = 100 and 1.579×10^{-15} for sparse & n = 1000 and 1.632×10^{-15} for dense & n = 1000 for n = 1000). We see the error of the dense method is 14.6% larger versus the sparse method for n = 100 and 3.4% larger for dense versus the sparse method for n= 1000. So the sparse method is more accurate than the dense method for both cases, and both methods are more accuracte for n = 100 than n = 1000 given this matrix. If the answer were to differ a lot between dense or sparse methods, it would grant further investigation.

Now looking at the accuracy of $\hat{x}$:

The condition number, which indicates sensitivity of the solution to changes in the input data, is fairly high at $1.3 \cdot 10^6$. Errors, even if starting off small, can compound when using iterative methods. Concluding, we should be cautious in general with solutions obtained with either dense or sparse methods when the matrix is ill-conditioned, especially for application where around 8 digits of accuracy is required in our estimate of x.