Astrophysical Lab: Winter Semester 2024 / 2025

Title of experiment: Linear Algebra - Matrix Inversion and The Rate Equations

Supervisor (1st afternoon): Prof. Meng Tian

Supervisor (2nd afternoon): Prof. Meng Tian

Group (number): 18

Names (of the students): Jennifer Fabà-Moreno (MA 1) and Pablo Vega del Castillo (MA 1)

(Please also indicate whether you are a Master's student in Astrophysics (MA) or in Physics (MP) and in which semester you are.)

1st afternoon (date, times of start and end): 13.30h - 17h

2nd afternoon (date, times of start and end): 13.30 - 17h

Time you needed to work on the report (at home) in person-hours, sum over all group members: 60 h

(This information does not enter into the evaluation, but it helps us to adjust the lab content.)

Here is space for your comments:

The professor explained us very well the manual as well as the jupyter notebook. Also told us about real aplications. It was really useful overall.

What did you like? How would you improve the labs? Would you actually like to implement these improvements yourself?

Signatures of the students, with date:

08/11/24

With your signatures you confirm that you have not used any references for your work other than those quoted in your lab report, and that you have not copied any content from other students' reports.

Evaluation of the lab report and the performance of the students during the lab (by the supervisor):

Signature of the lab supervisor, with date:

Numerical Laboratory
Linear Algebra - Matrix Inversion and The Rate Equations

Abstract. In this session we are going to deal with numerical matrix inversion. In particular we are going to work with numerical methods for solving systems of linear equations, which we will apply for finding the inverse of a matrix.
In Physics and, in particular, in Astrophysics, linear systems of equations (or linearized versions of them) are of the uttermost importance when describing complex systems. Assuming certain conditions, these linearized versions help us to find an approximate solution within this regime. Some examples are the linearized Einstein equations [1], which let us describe gravitational waves, or partial differential equations such as the heat equation or the diffusion equation.

Jennifer Fabà-Moreno
Pablo Vega del Castillo
Group 18

Prof. Meng Tian: meng.tian@physik.lmu.de

November 8, 2024

1 Theoretical framework

Searching for powerful algorithms which let us solve linear systems of equations will be our main objective. We will analyze two methods: Gaussian elimination and LU decomposition (refer to 'Matrix inversion' in [2] for an extended theoretical foundation). Given a matrix equation of the form $Ax = b$, where A and b are known, the Gaussian elimination process consists in a series of row operations to transform the so called coefficient matrix A into an upper triangular matrix. Once this is achieved we are able to solve for the entries of the column vector x using back substitution. Alternatively, the LU decomposition method is based on the premise that solving a matrix equation in which the A matrix is a triangular matrix requires less operations than if it was not. Based on that, the aim of this method as it name states is to find the lower triangular matrix L and the upper triangular matrix U such that $A = LU$. This way the original equation can be expressed as $LUx = b$ which allows us to solve the problem in two steps: $Ly = b$, which we solve for y using forward substitution, followed by $Ux = y$, which we solve for x using now backward substitution.

In some tasks, in order to check the validity of our algorithms, we are going to work with simple examples, which we can manually solve and then compare the solution with the one provided by our code. Then, we will apply our code to particular cases within the Astrophysical frame, such as the study of emission lines in planetary nebulae. In fact, we are going to study the dependence of the line ratios of two oxygen ions (OII and OIII) with density and temperature. We are going to focus in the scenario in which the density of the nebula is intermediate, since this will let us work with the algorithm for linear systems we have developed. Thus, the equations we have to solve for the population of each energy level come from a balance of the number of transitions into a level and those leaving it [2]. For energy state i:

$$n_i \sum P_j = \sum_j n_j P_{ji} \tag{1.1}$$

The transition rate P_{ij} takes into account the contributions from collisional $C_{ij} = n_e q_{ij}$ and radiative processes but the latter are only to be included in the downward rates (photon emission is the main mechanism through which ions lose energy).

On the other hand, we are going to implement benchmarking for the accuracy of our results. The philosophy of benchmarking [3] is based in the comparison and improvement of the techniques used for dealing with a task. It is not only related to computational ones but rather inherent to structuring and evaluating performance. In our case, we are going to work with different precisions and compare the required computational time and the error in the solution, in order to select the most efficient algorithm.

1.1 Comparison of computational orders of algorithms for matrix inversion

We can apply the above-mentioned techniques for solving linear systems to numerically compute the inverse of a given matrix $A \in M(\mathbb{R}_{n\times n})$.

$$AX = Id \iff \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \tag{1.2}$$

Considering vector notation, we can define:

$$X = (\mathbf{X_1}, \mathbf{X_2}, \cdots, \mathbf{X_n}) \qquad\qquad Id = (\mathbf{e_1}, \mathbf{e_2}, \cdots, \mathbf{e_n})$$

where each vector represent a column of the original matrix. Therefore $\{e_i\}_{i=1}^n$ are the canonical basis vectors of dimension n.

Now, we can rewrite Equation 1.2 as $A\mathbf{X_i} = \mathbf{e_i}$ for $i = 1, \cdots, n$. Thus, finding the expression of the inverse of A is equivalent to solving n systems of linear equations of the form:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} \mathbf{X}_{i1} \\ \mathbf{X}_{i2} \\ \vdots \\ \mathbf{X}_{in} \end{pmatrix} = \mathbf{e_i} \tag{1.3}$$

Let us now compare the required computational time for the two methods we use for solving linear systems:

- Gaussian elimination. This technique is of order $O(n^3)$ for a $n \times n$ matrix. Therefore, as we should apply it n times (one for each column of the inverse matrix), implementing this method for computing the inverse is of order $O(n^4)$.
- LU decomposition. This technique is also of order $O(n^3)$ for a $n \times n$ matrix. From Equation 1.3, we see that the matrix A is involved in the n systems of equations we must solve. Thus, we only need to perform the LU decomposition of A once, since we can store this result. Then, solving each of the n systems of equations takes a computational time of order $O(n^2)$, since we have the triangular form of A. Explicitly, the total computational time order for the LU decomposition method:

$$O(n^3) + nO(n^2) = O(n^3) + O(n^3) \approx O(n^3) \tag{1.4}$$

We conclude that LU decomposition is computationally more efficient than performing Gaussian elimination for finding the inverse of a matrix.

1.2 Tridiagonal matrices

We are working with this particular kind of matrices, since they are frequently used when working with recurrent systems of equations and partial differential equations solved by finite difference methods. The matrix of coefficients A (to be consistent with our notation) has non-zero elements only in the diagonal and in the above and below diagonals, hence the tridiagonal characterization. We can take advantage of this fact to refine the method for solving these systems using the Thomas/Progronki Method. It consists in the definition of some coefficients that will let us solve the system by simple back-substitution (refer to Matrix Inversion [2] for a more complete explanation). Thus, the required computational time for this method is $O(n)$ for a $n \times n$ matrix.

2 Tasks

2.1 Gaussian Elimination and LU decomposition

Exercise 1

- Manually solve the system of equations:

$$\begin{pmatrix} 2 & 2 & 3 & 1 \\ 3 & 4 & -2 & 5 \\ -5 & 5 & -1 & -2 \\ -1 & -1 & -3 & 3 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 3 \\ -2 \end{pmatrix} \tag{2.1}$$

using Gaussian Elimination without using Pivoting.

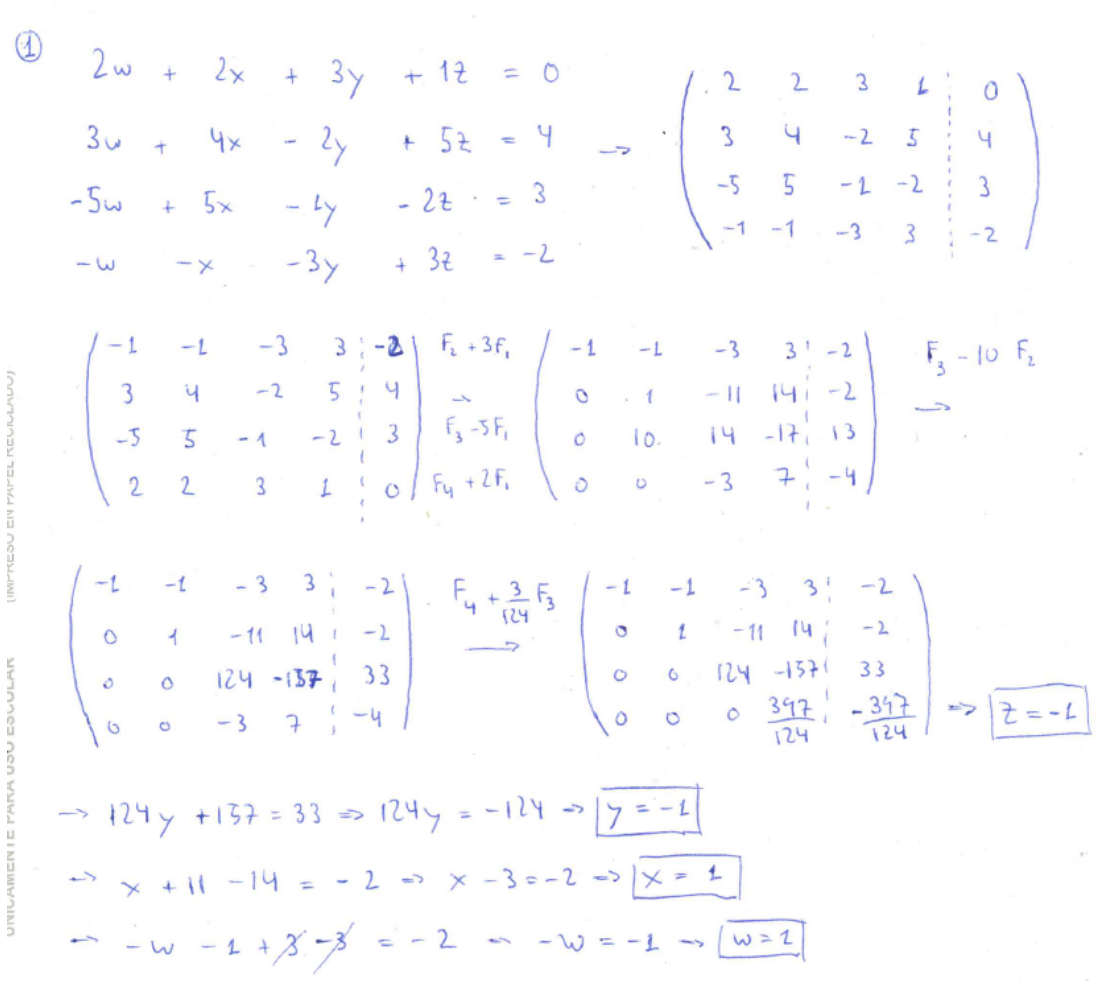

Figure 2.1: Manual solution of Equation 2.1.

- Solve the above equation this time using the LU decomposition using partial pivoting.
  Now, using LU decomposition with partial pivoting we follow the procedure as can be seen in Figure 2.2.

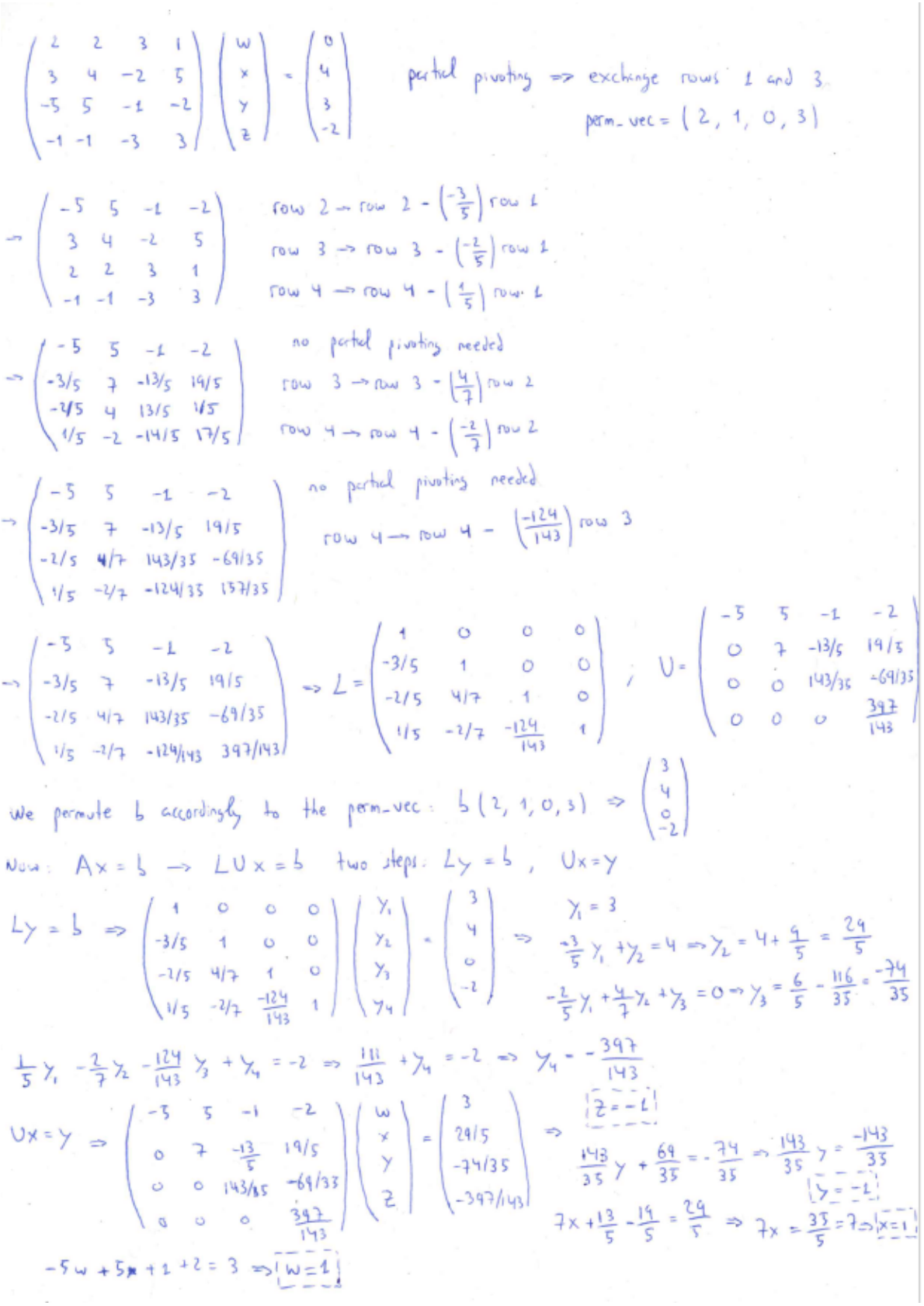

Figure 2.2: Manual solution with LU of Equation 2.1.

- Define the variable solution_vec in the following section with your solution. We will use this as a unit test for all following algorithms.
  As we have computed the result from Equation 2.1, we can complete the already defined function `TEST_INV` (the `solution_vec` variable) with our result. Thus, it is already functional for further testing with the algorithms we are going to create throughout the project.

```
def TEST_INV(fnkt_inv_mat):
    """ Simple routine to test the accuracy of
    the matrix inversion function my_inv.

    Parameters:
    -----------
    fnkt_inv_mat: function(np.array, RHS_vec), takes a numpy array and
                  a vector for the right hand side of the linear equation,
                  returns a numpy vector with the solution of the equation


    Returns:
    --------
    succ_flag: integer,
               succ_flag is 0 --> test failed
               succ_flag is not 0 --> test succeeded
    """

    #
    #test matrix:

    test_matrix = np.array([[2.0, 2.0, 3.0, 1.0],
                            [3.0, 4.0, -2.0, 5.0],
                            [-5.0, 5.0, -1.0, -2.0],
                            [-1.0, -1.0, -3.0, 3.0]])
    test_vec_RHS = np.array([0.0, 4.0, 3.0, -2.0])

    #your result:
    solution_vec = np.array([1., 1., -1., -1.])

    #threshold for numerical accuracy
    num_acc = 1e-6

    #TESTING BENCHMARK
    sol_routine = fnkt_inv_mat(test_matrix, test_vec_RHS)
    print(np.sum(np.abs(solution_vec - sol_routine)))
    if np.sum(np.abs(solution_vec - sol_routine)) > num_acc:
        print('Test failed')
        return 0
    else:
        print('Test successful')
        return 1
```

Listing 1: Test_Inv algorithm.

Exercise 2

- Write the implementation of the `noPivoting` method that serves us as a "dummy method" that disables pivoting.
  In this situation, as we do not have to perform pivoting when solving the system of linear equations, we want the function to take a matrix `mat` and an integer `col_idx` as arguments and return the element `col_idx` of the diagonal of the matrix `mat`.

```
def noPivoting(mat, col_idx):
    return col_idx, np.abs(mat[col_idx, col_idx]) #returns the absolute value
     of diagonal element of the column nre col_idx
```

Listing 2: noPivoting algorithm.

- Write the implementation of the `implicitPivoting` method that implements implicit pivoting.
  In this case, we want to implement the implicit pivoting method, which compares the normalized values of the elements in each column for each independently normalized row. Following the `partialPivoting` function, we define the implicit one as follows:

```
1 def implicitPivoting(mat, col_idx):
2     apivot = np.abs(mat[col_idx, col_idx])/np.sum(np.abs(mat[col_idx, :])) #
      We choose the normalized diagonal element as pivot as our first candidate
3     p = col_idx #The initial position of the pivot is taken as the diagonal
      element of the column
4     for l in range(col_idx+1, mat.shape[0]):
5         if np.abs(mat[l,col_idx])/np.sum(np.abs(mat[l, :])) > np.abs(apivot):
6             apivot = mat[l, col_idx]
7             p = l    #Both the pivot and the its position change if any
      element of the column is greater than the previous chosen pivot. The loop
      compares only elements below the diagonal.
8     return p, np.abs(apivot) #Returns the index position of the pivot (in
      column col_idx) as well as its absolute value.
```

Listing 3: implicitPivoting algorithm.

2.2 Implementing the Matrix Inversion

In the following code we have implemented the case where the input parameter `b` is `None`. Under this circumstance, we are going to consider that the user wants to find the inverse of an input matrix `a`, so `b` will be considered as the Identity matrix of the same dimension of `a`.
As stated in section 1, we are going to perform LU decomposition for solving each of the columns of the inverse matrix (lines $24 - 25$ in the code) using the already implemented function `LU` (see Appendix C).

```
1  def solve_eq(a, b=None, pivotMethod=partialPivoting):
2      """ Solve the equation system LU x = b
3      If b is None:
4      Matrix inversion: b is set to be the identity matrix
5
6      Parameters:
7      ----------
8      a: numpy array, input matrix
9      if b is not None:
10     b: numpy vector, right hand side of equation system
11     pivotMethod: function
12
13     Returns:
14     --------
15     x: numpy vector, solution x of the linear system of equations
16     """
17
18     LU_decomp = LU(a, pivotMethod)
19
20     if b is None:
21         n = a.shape[0]
22         b = np.eye(n) #identity matrix nxn
23         inverse = np.zeros((n,n))
24         for i in range(n): # loop through each column
25             inverse[:, i] = sol_LU(LU_decomp[0], b[:, i], LU_decomp[1])
26         return inverse
27     else:
28         return sol_LU(LU_decomp[0], b, LU_decomp[1])
```

Listing 4: Algorithm for matrix inversion and solution of a system of equations.

We apply the `TEST_INV` function for testing `solve_eq`:

```
#Eq system

TEST_INV(solve_eq)
-> output
0.0
Test successful
1
```

Listing 5: Testing solve_eq.

With this result, we have checked the validity of our function and, thus, we are now ready to use the function for further tests.[1]

2.3 Benchmarking the LU decomposition

Exercise 3

- Generate random matrices of varying size (10 - 100 is already enough). Apply the LU decomposition with and without pivoting and compare their error.

  We are going to study the influence of the matrix size in the error and required time for the computation of the inverse using the function `get_mat_err` (see Appendix C). As for now, we are focused only in this dependence, we will not vary the floating-point precision. We devote the next question to deeply analyze the influence of the precision in the results. As we are not specifying any floating point for the data, we check the precision in which the computer is working by using:

```
import sys
print(sys.float_info)

--> output
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min
    =2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig
    =53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Listing 6: Computer precision by default.

  So as it is working with 15 digits precision, the results of this section are in float64.

  We will use the 3 pivoting methods we have implemented to check whether they influence the results. As the floating precision in a computer is finite, we expect rounding errors to exist. Therefore, using partial or implicit pivoting may improve the accuracy of the inverse by reducing operations with numbers of different orders of magnitudes and avoiding divisions by very small numbers.

```
#Dimensions we are going to study
dimensions = np.array([10,15,20,30,40,50,60,75,100])

#Create matrices
matrices = {}
for n in dimensions:
    matrices[n] = np.random.uniform(-10, 10, (n,n))

#Initialise the error and the computational time vectors
errors_no = np.zeros(len(dimensions))
errors = np.zeros(len(dimensions))
errors_imp = np.zeros(len(dimensions))
```

[1] For testing the noPivoting method, we have changed the implementation of `TEST_INV` by adding the pivoting method in line 35. We obtain the same output.

```
13
14 times_no = np.zeros(len(dimensions))
15 times = np.zeros(len(dimensions))
16 times_imp = np.zeros(len(dimensions))
17
18 #Compute values
19 for i, n in enumerate(dimensions):
20     errors[i], times[i] = get_mat_err(matrices[n])
21     errors_no[i], times_no[i] = get_mat_err(matrices[n], noPivoting)
22     errors_imp[i], times_imp[i] = get_mat_err(matrices[n], implicitPivoting)
```

Listing 7: LU decomposition for random matrices.

We use the same code (code below) for obtaining Figure 2.3 and Figure 2.4 but changing the object of study (error or computational time).

```
1 #Linear regression is only used for computational time. For errors, we do not
      compare with a theoretical value.
2
3 #No pivoting regresion
4 model_no = LinearRegression()
5 model_no.fit(np.log(dimensions).reshape(-1,1), np.log(times_no))
6 y_no = model_no.predict(np.log(dimensions).reshape(-1,1))
7
8 #Partial pivoting regresion
9 model = LinearRegression()
10 model.fit(np.log(dimensions).reshape(-1,1), np.log(times))
11 y = model.predict(np.log(dimensions).reshape(-1,1))
12
13 #Implicit pivoting regresion
14 model_imp = LinearRegression()
15 model_imp.fit(np.log(dimensions).reshape(-1,1), np.log(times_imp))
16 y_imp = model_imp.predict(np.log(dimensions).reshape(-1,1))
17
18
19 slope_no = model_no.coef_[0]
20 slope = model.coef_[0]
21 slope_imp = model_imp.coef_[0]
22
23 print(f"Slope of regresion with no Pivoting: {slope_no}")
24 print(f"Slope of regresion with Partial Pivoting: {slope}")
25 print(f"Slope of regresion with Implicit Pivoting: {slope_imp}")
26
27
28 plt.figure(figsize=(7, 5))
29 plt.scatter(np.log(dimensions), np.log(times_no), label = 'No Pivoting')
30 plt.plot(np.log(dimensions), y_no, color='red', label='Regression Line No
      Pivoting')
31
32 plt.scatter(np.log(dimensions), np.log(times), color='green', label = '
      Partial Pivoting')
33 plt.plot(np.log(dimensions), y, color='orange', label='Regression Line
      Partial Pivoting')
34
35 plt.scatter(np.log(dimensions), np.log(times_imp), color='purple', label = '
      Implicit Pivoting')
36 plt.plot(np.log(dimensions), y_imp, color='yellow', label='Regression Line
      Implicit Pivoting')
37
38 plt.xlabel('log(matrix dimension)')
39 plt.ylabel('log(computational time for inversion')
40 plt.grid(False)
41 plt.legend()
```

```
42 plt.show()
```

Listing 8: Error and computational time as a function of matrix size. Linear regression for theoretical comparison.

In Figure 2.3, we find that the pivoting method that leads to a higher error is no pivoting. In this case, the algorithm performs LU decomposition without taking into account the value of the pivot, so it might be dividing by very small quantities. On the whole, partial and implicit pivoting have the same error on the computations. As the elements of the matrices we have created have the same order, there is little difference when normalizing them. We see that the errors range from 10^{-9} and 10^{-14}, so they expand over 5 orders of magnitude depending on the matrix size (which covers 2 orders of magnitude). We see the exponential behavior of the error with respect to the matrix size.

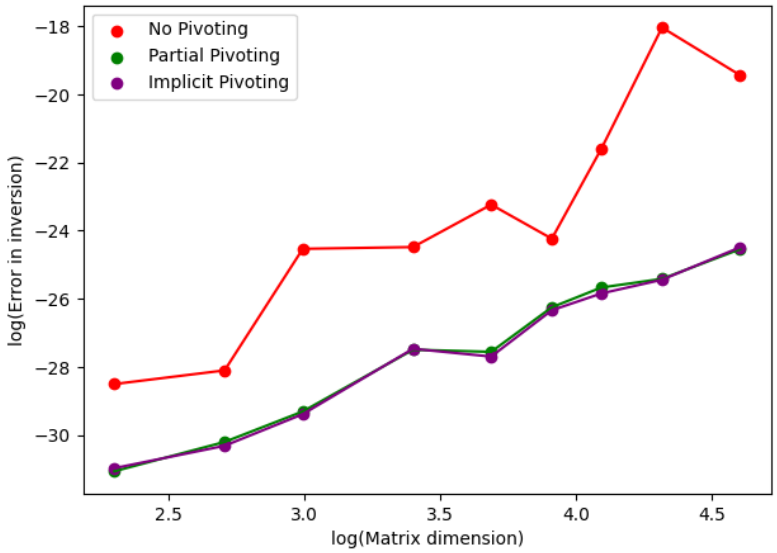

Figure 2.3: Error in inversion vs matrix dimension. Logarithmic scale.

- Plot the matrix size against the execution time in a log-log plot and compare the slope with the theoretical value of 3.

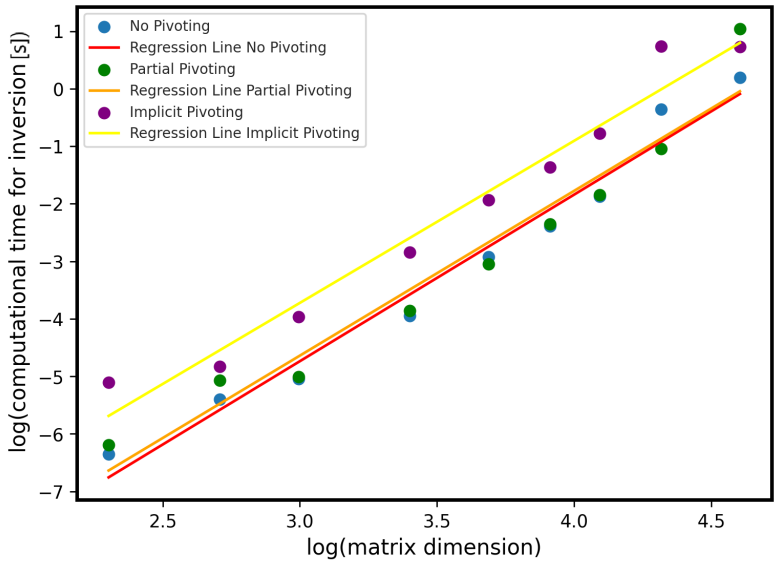

Figure 2.4: Computational time of inversion vs matrix dimension. Logarithmic scale.

Pivoting Method	Experimental Slope
no Pivoting	2.894603
Partial Pivoting	2.863071
Implicit Pivoting	2.816450

Table 2.1: Slope of the linear regression for the computational time of inversion vs matrix dimension. 6 decimals precision.

On the other hand, we fit a linear regression to the logarithm of the data computational time and matrix size, to compare it with the expected theoretical value of 3. This value comes from the exponent of n^3 of the computational order of the LU decomposition (see section 1) as we are working in log scale. From Table 2.1, we see that the 3 slopes approach 3. We also notice that the computational time for the implicit pivoting method is the highest. This is something reasonable as the implicit pivoting requires the normalization of the elements below the diagonal for each iteration.
Overall this extra computational time of the implicit pivoting method is balanced by its lower error with respect to the other methods.

- Why is the execution time with 16 bit slower than with 64 bit?

```
    # Floating point precission

n_size = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

float16_no = get_bench(np.float16, n_size, pivotMethod=noPivoting)
float32_no = get_bench(np.float32, n_size, pivotMethod=noPivoting)
float64_no = get_bench(np.float64, n_size, pivotMethod=noPivoting)

float16_part = get_bench(np.float16, n_size, pivotMethod=partialPivoting)
float32_part = get_bench(np.float32, n_size, pivotMethod=partialPivoting)
float64_part = get_bench(np.float64, n_size, pivotMethod=partialPivoting)

float16_imp = get_bench(np.float16, n_size, pivotMethod=implicitPivoting)
float32_imp = get_bench(np.float32, n_size, pivotMethod=implicitPivoting)
float64_imp = get_bench(np.float64, n_size, pivotMethod=implicitPivoting)

fig = plt.figure()

plt.scatter(np.log(float16_no[:,0]), np.log(float16_no[:,2]), label='float16, noPivoting')
plt.plot(np.log(float16_no[:,0]), np.log(float16_no[:,2]))
plt.scatter(np.log(float32_no[:,0]), np.log(float32_no[:,2]), label='float32, noPivoting')
plt.plot(np.log(float32_no[:,0]), np.log(float32_no[:,2]))
plt.scatter(np.log(float64_no[:,0]), np.log(float64_no[:,2]), label='float64, noPivoting')
plt.plot(np.log(float64_no[:,0]), np.log(float64_no[:,2]))

plt.scatter(np.log(float16_part[:,0]), np.log(float16_part[:,2]), label='float16, partialPivoting')
plt.plot(np.log(float16_part[:,0]), np.log(float16_part[:,2]))
plt.scatter(np.log(float32_part[:,0]), np.log(float32_part[:,2]), label='float32, partialPivoting')
plt.plot(np.log(float32_part[:,0]), np.log(float32_part[:,2]))
plt.scatter(np.log(float64_part[:,0]), np.log(float64_part[:,2]), label='float64, partialPivoting')
plt.plot(np.log(float64_part[:,0]), np.log(float64_part[:,2]))

plt.scatter(np.log(float16_imp[:,0]), np.log(float16_imp[:,2]), label='float16, implicitPivoting')
```

```
plt.plot(np.log(float16_imp[:,0]), np.log(float16_imp[:,2]))
plt.scatter(np.log(float32_imp[:,0]), np.log(float32_imp[:,2]), label='
    float32, implicitPivoting')
plt.plot(np.log(float32_imp[:,0]), np.log(float32_imp[:,2]))
plt.scatter(np.log(float64_imp[:,0]), np.log(float64_imp[:,2]), label='
    float64, implicitPivoting')
plt.plot(np.log(float64_imp[:,0]), np.log(float64_imp[:,2]))

plt.xlabel('log(matrix dimension)')
plt.ylabel('log(computational time [s])')
plt.legend()
#fig.savefig('floating_point_time_log.png')
plt.show()
```

Listing 9: Computing and plotting of the computational time vs. matix size for the different float types and the different pivoting methods.

Although the question already states that the computational time for inversion is higher when working with precision 16 than with precision 64, we can observe this phenomenon in Figure 2.5.

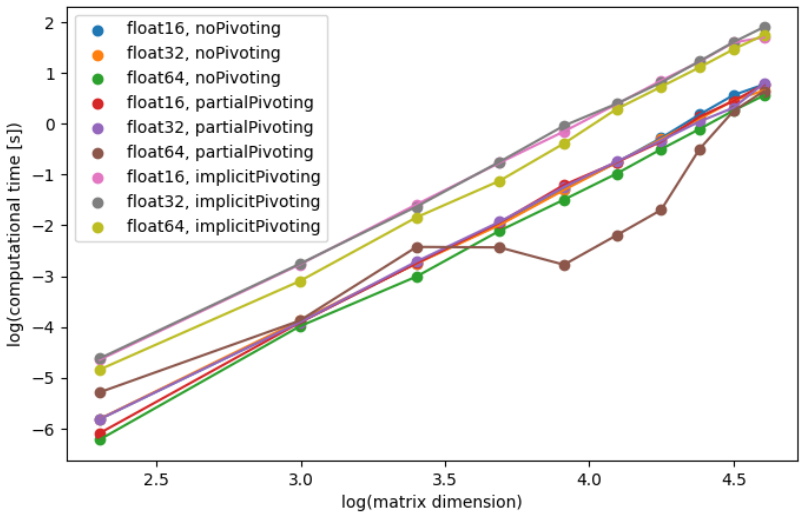

Figure 2.5: Computational time vs matrix size for the different float types.

While this could seem a little unreasonable, considering that if we are working with less precision we need less memory so the computations should be faster, our main guess for this to happen is that current computers are fitted for working with 64-double precision rather than with float16 [4]. Thus, when introducing float16 numbers the computer has to convert them or work in some way with them to be able to do the computations which ends up requiring more computational time. In Figure 2.5 we also observe that the implicit pivoting method is the one that takes the longest in accordance with was already seen in Figure 2.4 while partial and no pivoting take up similar time.

Additionally, we look now at the produced error when working with the three different float types as can be seen in Figure 2.6. It is worth mentioning that float16 type always leads to large errors that are even greater than the actual entries of the matrix. We therefore conclude that working with this float type is an useless approach. On the other hand the float64 type carries much smaller errors whose order is expected considering that we are working with numerical methods.

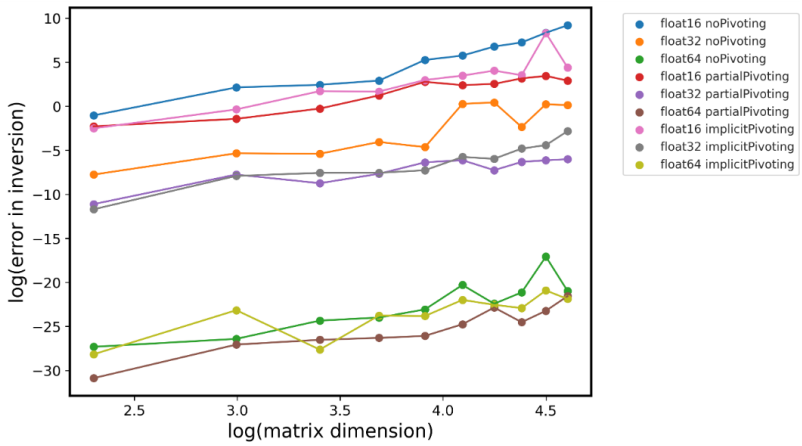

Figure 2.6: Error made in inversion for different float types and different pivoting methods.

Exercise 4. Hilbert Matrices

- Generate random matrices of varying size (3 - 50 is already enough)
- Apply the LU decomposition with and without pivoting and compare their error

We are going to work with an special type of matrices, which are the Hilbert matrices. Their elements are completely characterized by the size of the matrix (see [2]).
They are really ill-conditioned objects (the determinant is really dependent of the matrix size), so we are going to analyze the effect of this parameter on the error when performing LU decomposition.

```
H = hilbert(10)
np.min(H)
np.max(H)

--> output
0.05263157894736842
1.0
```

Listing 10: Hilbert matrix of dimensionn 10.

We compute a 10×10 Hilbert matrix just to see their explicit elements. You can find the matrix in Appendix B. We see that its elements vary considerable and span around 10 orders of magnitudes. This let us see the complications of working with them in numerical experiments.

```
dimensions = np.random.randint(3, 50, 15)
hilbert_results_no = get_bench_hilbert(np.float64, dimensions, noPivoting)
hilbert_results_imp = get_bench_hilbert(np.float64, dimensions, implicitPivoting)
hilbert_results = get_bench_hilbert(np.float64, dimensions, partialPivoting)

sort_indices = np.argsort(dimensions)
sort_dimensions = dimensions[sort_indices]
sort_errors = hilbert_results[:,2][sort_indices]
sort_errors_no = hilbert_results_no[:,2][sort_indices]
sort_errors_imp = hilbert_results_imp[:,2][sort_indices]

#expect the error to blow up at low n (size of the matrix)
plt.figure(figsize=(7, 5))
plt.scatter(sort_dimensions, sort_errors, color='blue', label='Partial Pivoting')
plt.plot(sort_dimensions, sort_errors, color='blue')
plt.scatter(sort_dimensions, sort_errors_imp, color='orange', label='Implicit
    Pivoting')
plt.plot(sort_dimensions, sort_errors_imp, color='orange')
plt.scatter(sort_dimensions, sort_errors_no, color='green', label='No Pivoting')
plt.plot(sort_dimensions, sort_errors_no, color='green')

```

```
plt.xlabel('Matrix dimension')
plt.ylabel('Error in inversion')
plt.legend()
plt.show()
```

Listing 11: Hilbert matrices algorithm. Precision: float64. Object of study: error.

From Figure 2.7, we see how the error increases exponentially with the dimension of the matrix. From low dimensions (p.e., let us say 20), the error escalates significantly, ultimately reaching values which are (at least) two orders of magnitude higher than the initial value (the minimum value is $\sim 1e-03$).

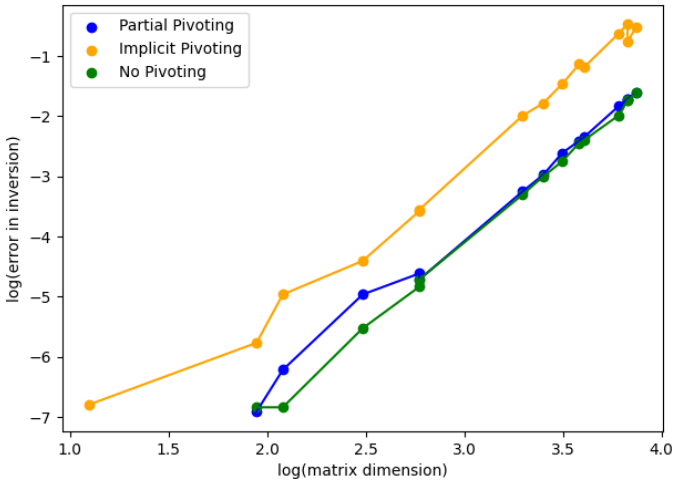

Figure 2.7: Error in inversion vs matrix dimension in Hilbert matrices.

When comparing different methods, we see that partial and no pivoting provide the same error (qualitatively). This was expected since the value of the determinant of these matrices varies significantly in order of magnitude when increasing the matrix dimension, so pivoting is not powerful enough to overcome this.
However, when comparing these both methods with the implicit pivoting one, we see a huge increase of the error. We consider that this could be due to the normalization of the elements of each row, since this could lead to make them even more different and, particularly, the ones of lower order would be also reduced to lower orders. On the other hand, partial pivoting, searches the greatest value and swap rows if needed but, at first, it does not affect the value of all the elements in the row (it does not divide by anything when comparing). Thus, as round-off errors are so significant when working with these matrices, implicit pivoting happens to be less effective in handling them.

We note that a message appears when running the code: lrloes_gb is not applicable. This is warning us about the fact that the determinant is 0 (numerically) so that the matrix is singular and is not supposed to have inverse (see function `LU` in Appendix C). This way, the algorithm is useful for making us notice that it fails at performing well.

2.4 Tridiagonal Matrices and the Thomas/Progronki Method

Exercise 5/6

- Manually solve the following system of equations by applying the Thomas/Progronki method.

$$\begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} \tag{2.2}$$

Tridiagonal matrices. thomas Progronki method.

Let us solve the following system of linear eq using the thomas algorithm, since we have a 3diagonal matrix:

$$\begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix} \rightarrow \begin{aligned} \{e_i\} &= -1 \quad i = 1, \dots, 4 \\ \{d_i\} &= 2 \quad i = 1, \dots, 5 \\ \{c_i\} &= -1 \quad i = 1, \dots, 4 \\ \{b_i\} &= i \quad i = 1, \dots, 5 \end{aligned}$$

We have $n = 5$. Let us compute α_i, β_i:

$$\alpha_1 = -1/2 = -\frac{1}{2}$$

$$\alpha_2 = \frac{-1}{2 - (-1)(-1/2)} = \frac{-1}{2 - 1/2} = -\frac{2}{3}$$

$$\alpha_3 = \frac{-1}{2 - (-1)\cdot(-2/3)} = \frac{-1}{2 - 2/3} = -\frac{3}{4}$$

$$\alpha_4 = \frac{-1}{2 - (-1)(-3/4)} = \frac{-1}{2 - 3/4} = -\frac{4}{5}$$

$$\beta_1 = \frac{1}{2}$$

$$\beta_2 = \frac{2 + (-1)\,1/2}{2 - (-1)(-1/2)} = \frac{2 - 1/2}{2 - 1/2} = 1$$

$$\beta_3 = \frac{3 + (-1)\cdot 1}{2 - (-1)(-2/3)} = \frac{3-1}{2 - 2/3} = \frac{2}{4/3} = \frac{6}{4} = \frac{3}{2}$$

$$\beta_4 = \frac{4 + (-1)\cdot 3/2}{2 - (-1)(-3/4)} = \frac{4 - 3/2}{2 - 3/4} = \frac{5/2}{5/4} = 2$$

$$\beta_5 = \frac{5 + (-1)\cdot 2}{2 - (-1)\cdot(-4/5)} = \frac{5-2}{2 - 4/5} = \frac{3}{6/5} = \frac{15}{6} = \frac{5}{2}$$

From here we can compute the value of x_i:

$$x_5 = \beta_6 = 2.5$$

$$x_4 = -\frac{4}{5}\cdot 2.5 + 2 = -2 + 2 = 0$$

$$x_3 = -\frac{3}{4}\cdot 0 + 3/2 = 3/2 = 1.5$$

$$x_2 = -\frac{2}{3}\cdot\frac{3}{2} + 1 = 0$$

$$x_1 = -\frac{1}{2}\cdot 0 + \frac{1}{2} = \frac{1}{2} = 0.5$$

the solution to the system is $(x_1, x_2, x_3, x_4, x_5) = (0.5, 0, 1.5, 0, 2.5)$

Figure 2.8: Manual solution of Equation 2.2

- Write a function to implement the Thomas/Progronki method in the following section.

```
def thomas(c, d, e, b):
    """ Thomas algorithm to solve the trigonal system of equations.

    Parameters:
    -----------
    c: numpy array, lower diagonal element of the trigonal matrix
    d: numpy array, diagonal element of the trigonal matrix
    e: numpy array, upper diagonal element of the trigonal matrix
    b: numpy array, right hand side of the system of equations

    Returns:
    --------
    x: numpy array, solution to the trigonal system of equations
    """

    #prefactor convention
    e = -e
    c = -c
    c = np.insert(c,0,0)   #the vector c does not have component c[0], so we
    add a 0 so that c[1] actually corresponds to the first element of c.

    #e does not have the element e[n-1] but this will not beused in our loops
    , so we do not add anything to it.
    n = len(d)

    alpha = np.zeros(n-1)
    beta = np.zeros(n)
    x = np.zeros(n)

    for i in range(n-1):
        if i == 0:
            alpha[i] = e[i]/d[i]
        else:
            alpha[i] = e[i]/(d[i]-c[i]*alpha[i-1])

    for i in range(n):
        if i == 0:
            beta[i] = b[i]/d[i]
        else:
            beta[i] = (b[i]+c[i]*beta[i-1])/(d[i]-c[i]*alpha[i-1])


    for i in reversed(range(0, n)):
        if i == (n-1):
            x[i] = beta[i]
        else:
            x[i] = alpha[i]*x[i+1]+beta[i]
    return x
```

Listing 12: Thomas Method algorithm.

- Compare the result you obtained manually with the output from your function.
  We are going to solve 2.2 with this algorithm and compare with the manually obtained value.

```
c = np.array([1.0,1.0,1.0,1.0])
d = np.array([2.0,2.0,2.0,2.0,2.0])
e = np.array([1.0,1.0,1.0,1.0])
b = np.array([1.0,2.0,3.0,4.0,5.0])
thomas(c,d,e,b)
-> output
array([5.00000000e-01, 2.22044605e-16, 1.50000000e+00, 0.00000000e+00,
       2.50000000e+00])
```

Listing 13: Thomas Method algorithm application.

We see that both approaches lead to the same solution (although in the latter, we have to take into account the finite precision of the machine).

Additionally, since we are implementing this method for reducing the computational time required for solving the system, we are going to explicitly compute the time that the computer takes to perform both approaches:

```
#Thomas
start_time = time.time()
x = thomas(c,d,e,b)
diff_time = time.time() - start_time
print(x, diff_time)

-> output
[5.00000000e-01 2.22044605e-16 1.50000000e+00 0.00000000e+00
 2.50000000e+00] 0.0


#LU
mat = np.array
    ([[2.0,1.0,0.0,0.0,0.0],[1.0,2.0,1.0,0.0,0.0],[0.0,1.0,2.0,1.0,0.0],
[0.0,0.0,1.0,2.0,1.0],[0.0,0.0,0.0,1.0,2.0]])
a = np.array([1.0,2.0,3.0,4.0,5.0])
start_time = time.time()
x = solve_eq(mat,a)
diff_time = time.time() - start_time
print(x, diff_time)
-> output
[5.00000000e-01 1.48029737e-16 1.50000000e+00 0.00000000e+00
 2.50000000e+00] 0.0009598731994628906
```

Listing 14: Computational time for LU decomposition and Thomas algorithm for 3-diagonal matrices.

As we see, the computational time required for doing the LU decomposition with partial pivoting is higher than the one for the Thomas method, which in fact is so little that the precision of the computer returns 0.0. We have experimentally proved the theoretical expectations.

2.5 Line ratios of OII and OIII

Exercise 7.

- Use the routines provided for a number of temperatures (O II) and densities (O III) to produce line ratios and plot them.
- Comment on your results.

Now we focus on the astrophysical application of the methods developed in the previous sections. We are going to compute the line ratios for both ions and study their dependencies with temperature and electron number density.
First we study the OII ion line ratios, which according to the theory are density-dependent. In Figure 2.9 we represent these ion line ratios against the electron number density for a range of temperature values $T \in [5000, 40000]K$. We can clearly observe density dependence on the values of the line ratios. However, when looking at the lowest densities of the plot, this dependence seems to vanish. This is in accordance with the theory since the electron density disappears from the expression of the line ratios. We can also extrapolate this behavior for the highest densities, which was also expected.

On the other hand, we see that there is little temperature dependence, since the Boltzmann factor in the line ratio expression is almost always 1, specifically, it is $e^{-\frac{20}{kT}}$ [2]. However, we are to

distinguish that for a fixed number density value the line ratios slightly increase with temperature following this exponential trend.

```
NDEN = 15
temperatures = [5000, 10000, 20000, 30000, 40000]  # Array ot temperature values from 5000 to 40000 K


line_ratios = {temp: np.zeros(NDEN) for temp in temperatures}
deltn = np.emath.log10(2.e0)
logne = np.linspace(deltn, NDEN * deltn, NDEN).tolist()

for t in temperatures: # loop over all temperatures
    for i in range(NDEN): # loop over all densities
        a = np.zeros((nlev_oii, nlev_oii))
        b = np.zeros(nlev_oii)
        ne = 10.e0 ** logne[i]
        setup_rates(nlev_oii, weight_oii, energy_oii, ups_oii, aij_oii, ne, t, a, b)
        b = solve_eq(a, b)

        # Calculate line ratio
        # The line ratio corresponds to the diagram given in the text.
        # Here we compare the number of transitions in the 1-2 line with the number in the 1-3 line.
        line_ratios[t][i] = b[1] * aij_oii[0, 1] / (b[2] * aij_oii[0, 2])

fig = plt.figure()
for t in temperatures:
    plt.scatter(logne, line_ratios[t], label=f'T = {t} K')
    plt.plot(logne, line_ratios[t])

plt.xlabel('log(number density [cm$^{-3}$])')
plt.ylabel('Line ratio')
plt.legend()
#plt.title('Line Ratio O II vs. Log Number Density for Different Temperatures')
fig.savefig('line_ratio_oii')
plt.show()
```

Listing 15: Code for computing the line ratios of the OII atom.

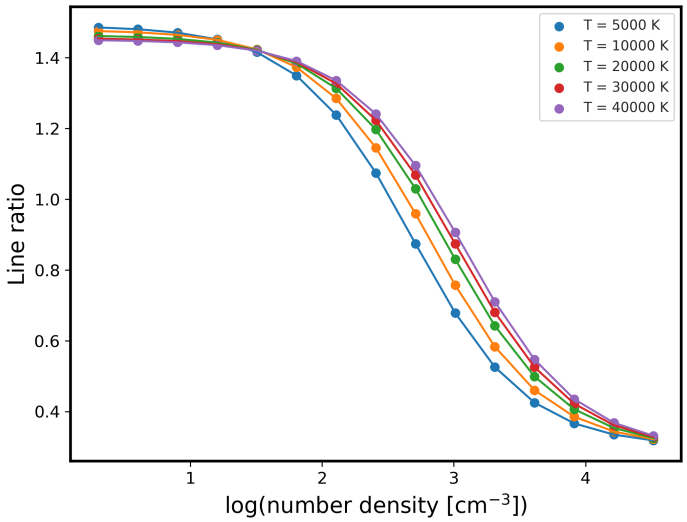

Figure 2.9: Line ratios for the O II atom vs. log(electron number density (n_e)) for different values of the temperature in the range 5000 - 40000 K.

Now we look at the OIII ion line ratios which have are plotted in Figure 2.10. In this case, we expect a dominance of the Boltzmann exponential term which means that the line ratios are temperature-dependent. We see that the pattern is similar to a $e^{1/x}$, which make us think that we are working with the inverse line ratio, since neither k neither T are negative. A small density dependence can be observed specially for lower temperature values, although as stated is the temperature factor the dominant one.

```
NTEMP = 15
densities = [10., 100., 1000., 10000., 100000., 1000000.]  # Array of densities
line_ratios = {ne: np.zeros(NTEMP) for ne in densities}  # Dictionary to store line ratios for each density
temp = np.linspace(6000, 6000 + (NTEMP - 1) * 1000, NTEMP)  # Temperature range from 6000K to 21000K

for ne in densities:
    for i in range(NTEMP):
        a = np.zeros((nlev_oiii, nlev_oiii))
        b = np.zeros(nlev_oiii)
        t = temp[i]
        setup_rates(nlev_oiii, weight_oiii, energy_oiii, ups_oiii, aij_oiii, ne, t, a, b)
        b = solve_eq(a, b)

        # Calculate line ratio
        line_ratios[ne][i] = b[3] * (aij_oiii[2, 3] + aij_oiii[1, 3]) / (b[4] * aij_oiii[3, 4])

fig = plt.figure()
for ne in densities:
    plt.scatter(temp, line_ratios[ne], label = f'n_e = {ne:.0e} cm^{-3}')
    plt.plot(temp, line_ratios[ne])

plt.xlabel('Temperature [K]')
plt.ylabel('Line Ratio')
plt.legend()
#plt.title('Line Ratio vs. Temperature for Different Densities')
fig.savefig('line_ratio_oiii')
plt.show()
```

Listing 16: Code for computing the line ratios for the OIII atom.

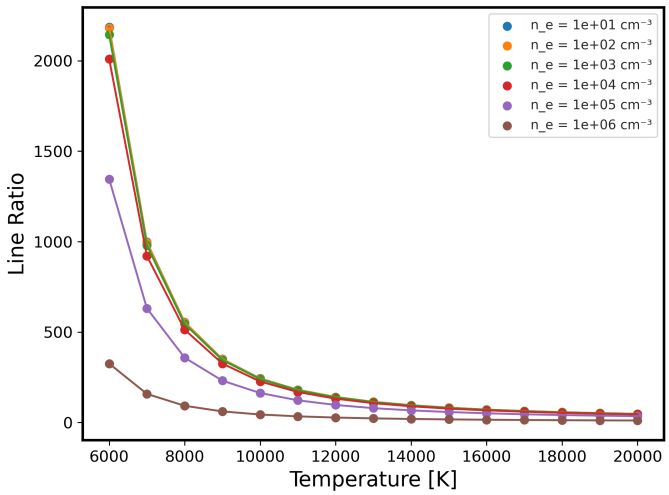

Figure 2.10: Line ratios for the O III atom vs. temperature for different values of the density in the range 10 - 10^6 cm^{-3}.

3 Advanced Tasks

Exercise 1. Iterative improvement of the solution

In this task, we have to implement an algorithm to improve the solution of the linear system. We suppose that the given solution x_0 is equal to the real one x but with a certain error δx_0 that is due to computational approximations:

$$x = x_0 + \delta x_0 \tag{3.1}$$

Now, we can solve the system:

$$A(x_0 + \delta x_0) = b \implies A\delta x_0 = b - Ax_0 \equiv r_0 \tag{3.2}$$

with LU decomposition where the vector in the rhs of the equation is r_0 and the solution is δx_0.

We can continue this process and consider that $x_0 = x_1 + \delta x_1$ so that $x = x_1 + \delta x_1 + \delta x_0$. We can iterate this up to n so that $x = x_n + \sum_{i=1}^{n} \delta x_i$. We decide to stop the algorithm when r_n<prec, for a certain chosen precision (following the criterion of `TEST_INV` function, since this means that the difference between the lhs and the rhs of the equation is small.

```
def iterative_sol(A, b, pivotMethod=partialPivoting):
    x = solve_eq(A,b,pivotMethod)
    prec = 1e-13 #1e-12 for Hilbert matrix
    r = np.ones(np.shape(b)[0])
    x1 = x.copy()
    i=0
    while np.sum(np.abs(r)) > prec:
        r = b-np.dot(A,x)
        x = solve_eq(A, r, pivotMethod)
        x1 = x1 + x
        i = i + 1
    return x1, i #we want to return the solution as well as the number of times
   that it has consider necessary to add a perturbation to the solution
```

Listing 17: Iterative improvement algorithm.

We are going to test our algorithm with a random matrix of size 20×20 and with a Hilbert matrix of dimension 5×5. We have asked the function to return both the result and the times it has needed to refine the result. We have set the precision to be $1e-13$ ($1e-12$ for Hilbert), since with higher precision the algorithm indefinitely entered the loop, in fact, we had to stop it from running. However, this leads the function to enter only once in the loop, which means that there is no need to improve the first computed solution. We have tried several times with different matrices and orders of the Hilbert one, but we obtain the same results.

```
test_matrix = np.random.uniform(-200, 200, (20,20))
test_vec_RHS = np.random.uniform(-1, 1, (20))
iterative_sol(test_matrix, test_vec_RHS, pivotMethod=partialPivoting)
--> output
(array([-0.00430087, -0.05390137,  0.01824677,  0.00787443,  0.00529454,
        -0.00480155, -0.01478623, -0.00044252, -0.01617922, -0.0012663 ,
        -0.03261902, -0.03555607, -0.0022594 , -0.02322338, -0.04208783,
        -0.01346007, -0.02173747, -0.01626085, -0.00635663,  0.02445687]),
 1)

#Hilbert
test_vec_RHS = np.random.uniform(-1, 1, (5))
iterative_sol(np.array(hilbert(5)), test_vec_RHS, pivotMethod=partialPivoting)
--> output
(array([  27.85953267, -148.38448691, -149.80802027,  870.53158746,
        -608.96476616]),
 1)
```

Listing 18: Iterative improvement test.

Exercise 2. Line ratios of SII and SIII

We now aim to calculate the line ratios for the SII and SIII ions using the code already implemented for the OII and OIII. For this task we display the necessary input data on the following code snippets.

```
###  Define the model S II atom. Compare these numbers with those given in the Appendix.
###  Number of levels, a label, statistical weight and energy (in cm^{-1})
nlev_sii=5
lab_sii=np.char.array(["4S3_2","2D3_2","2D5_2","2P1_2","2P3_2"])
weight_sii=np.array([4,4,6,2,4])
energy_sii=np.array([0.0,14852.94,14884.73,24524.83,24571.54])
### With n levels there are n*(n-1)/2 possible transitions.
###  Upsilon is dimensionless while A has dimension sec^{-1}.
ups_sii=np.array([[0.,2.79,4.19,0.76,1.52],
             [0.,0.,7.59,1.52,3.38],
             [0.,0.,0.,2.56,4.79],
             [0.,0.,0.,0.,2.38]])
aij_sii=np.array([[0.,8.8e-4,2.6e-4,9.1e-2,2.2e-1],
              [0.,0.,3.3e-7,1.6e-1,1.3e-1],
              [0.,0.,0.,7.8e-2,1.8e-1],
              [0.,0.,0.,0.,1.0e-6]])
```

Listing 19: Input data for obtaining the SII line ratios

```
    ###  Define the model S III atom.  Again, check these numbers with those listed in the Appendix
###  Number of levels, a label, statistical weight and energy (in cm^{-1})
nlev_siii=5
lab_siii=np.char.array(["3P0","3P1","3P2","1D2","1S0"])
weight_siii=np.array([1,3,5,5,1])
energy_siii=np.array([0.0,298.69,833.08,11322.7,27161.0])
### With n levels there are n*(n-1)/2 possible transitions.
###  Upsilon is dimensionless while A has dimension sec^{-1}.
ups_siii=np.array([[0.,2.64,1.11,0.93,0.13],
             [0.,0.,5.79,2.80,0.40],
             [0.,0.,0.,4.66,0.66],
             [0.,0.,0.,0.,1.88]])
aij_siii=np.array([[0.,4.7e-4,4.6e-8,5.8e-6,0.0],
              [0.,0.,2.1e-3,2.2e-2,8.0e-1],
              [0.,0.,0.,5.8e-2,1.0e-2],
              [0.,0.,0.,0.,2.2]])
```

Listing 20: Input data for obtaining the SIII line ratios

In the case of the SII atom we are able to see from Figure 3.1 the dependence of the line ratio with the electron number density. As in the OII case, for lower densities this dependence vanishes and the line ratio is approximately constant. However, comparing this graph with the one of the OII, Figure 2.9, reveals that the SII line ratio follows the inverse trend: it increases with the number density as opposed to the OII scenario. To explain this we have to notice that the line ratio which is computed for the SII atom is different from the one of the OII atom. For the OII atom:

$$line\ ratio(OII) = \frac{1 \to 0}{2 \to 0} = \frac{^2D_{5/2} \ \to \ ^4S_{3/2}}{^2D_{3/2} \ \to \ ^4S_{3/2}}$$

while for the SII, the energy levels $^2D_{5/2}$ and $^2D_{3/2}$ are swapped so:

$$line\ ratio(SII) = \frac{1 \to 0}{2 \to 0} = \frac{^2D_{3/2} \ \to \ ^4S_{3/2}}{^2D_{5/2} \ \to \ ^4S_{3/2}}$$

This means that the position of the energy levels of the atom affects how the line ratio behaves against the electron number density.

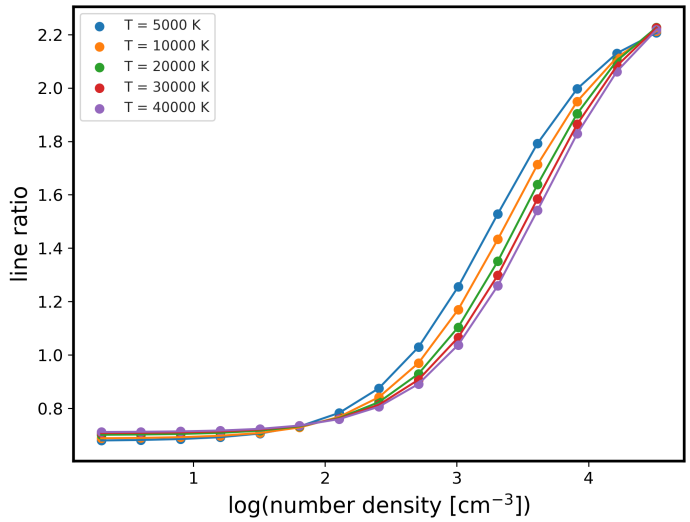

Figure 3.1: Line ratios for the SII atom.

In the case of the SIII atom we see that the line ratio depends on the temperature in a very similar way as the OIII atom. In fact, the SIII atom maintains the same disposition in the energy levels as the OIII atom so this similarity in the graphs is thus explained.

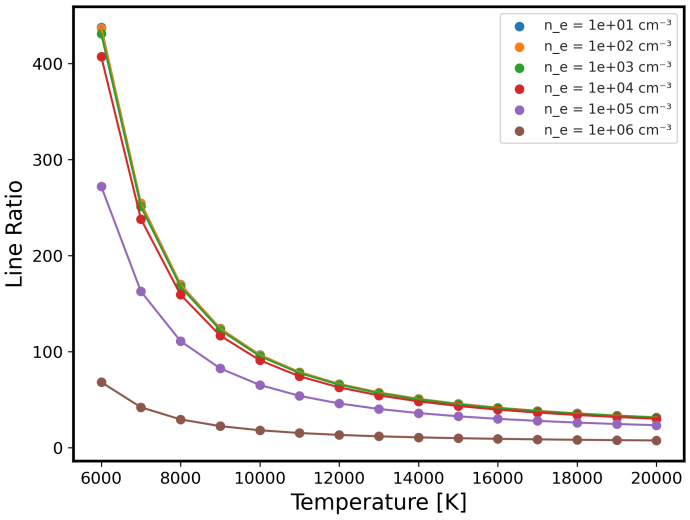

Figure 3.2: Line ratios for the SIII atom.

4 Conclusions

We have conducted a detail analysis of the numerical method of matrix inversion by solving linear systems of equations. Although this may appear straightforward, it is crucial for a wide array of applications in physics. Physicists often seek to explain natural phenomena by modeling systems under simplified conditions, which usually results in linear systems of equations.

First, we implemented Python code to compute both the inverse of a matrix and the solution to a system of linear equations within the function `solve_eq`. We considered 3 different LU methods for doing so: no pivoting, partial and implicit pivoting. To test the function's efficiency and accuracy, we utilized the pre-existing `TEST_INV` function, which applies `solve_eq` to a test matrix and vector solution. This test function confirms success if the Euclidean distance between the theoretical solution and our numerical result is lower than a specified precision.
Since this test was successful, we used our function in random matrices to evaluate its performance. By using the functions `get_mat_err` and `get_bench` we obtained the error and required computational time and were able to analyzed their dependence with the matrix size. In general, we conclude that the no pivoting method is the least accurate. In fact, errors between this method and the pivoting ones differ generally differ by 2 orders of magnitudes. Regarding the computational time, we observed that the floating precision plays a crucial role. Since most of the current software are adapted to handle with float64 rather than with float16, they take extra time when doing computations with this kind of objects.

Then, we apply our tested code in 2 different types of matrices, as well as in a particular case: the study of 2 ions of oxygen. Regarding the former mentioned, we study the ill condition of the Hilbert matrices, where we see that the error increased exponentially with the matrix dimension, independently of the pivoting method used. On the other hand, we implemented the Thomas/Progronki method for solving tridiagonal matrices, since it is (both theoretically and experimentally) more efficient than LU decomposition.
Regarding the latter mentioned, we applied our implemented numerical methods to the particular case of studying line ratios in the energy levels of the OII and OIII ions. We have demonstrated then the utility of these methods by obtaining the desired dependencies in the line ratios with temperature and electron number density as stated by theory. Therefore these tools are an efficient approach to the study of the composition and abundance of elements present in planetary nebulae and can be a starting point in the analysis of the spectra of these astronomical objects.

In conclusion, we have evidenced the significance of selecting an appropriate algorithm for solving systems of equations. In numerical experiments and simulations, thorough data analysis and benchmarking are essential for achieving accurate results, as taking into account symmetries and other specific characteristics of the data (which leads to simplify the systems we have to solve) can greatly enhance computational efficiency and precision.

5 Author's note

We are aware of the maximum extension of the report (12 text pages). Hence when considering only the written part of this report (not taking into account neither codes nor figures), it is about 10 pages.

Bibliography

[1] Steven Weinberg. "Gravitation and cosmology: principles and applications of the general theory of relativity". In: (1972) (cit. on p. 1).

[2] Tadziu Hoffmann Keith Butler. *Computational Methods in Astrophysics. Linear Algebra — Matrix Inversion and The Rate Equations*. https://www.usm.lmu.de/Lehre/Lehrveranstaltungen/Praktikum/anleitungen.php. Accessed: Oct, 2024 (cit. on pp. 2, 3, 13, 17, 26).

[3] Inc. Wikimedia Foundation. *Benchmarking*. https://en.wikipedia.org/wiki/Benchmarking. Accessed: Oct, 2024 (cit. on p. 2).

[4] Python Software Foundation. *Floating-Point Arithmetic: Issues and Limitations*. https://docs.python.org/3/tutorial/floatingpoint.html. Accessed: Nov, 2024 (cit. on p. 12).

[5] Maria Rosa Camps Camprubí. "Introducció a l'àlgebra lineal". In: (2005) (cit. on p. 25).

A Determinant of the product of 2 matrices

A necessary and sufficient condition for a matrix to be non-singular (that is, invertible) is that its determinant is not 0. Thus, before applying our algorithm to compute the inverse of a given matrix, we should check the value of its determinant.

This section aims to prove that the determinant of a given matrix can be computed as the product of the elements in the diagonal of U (from the LU decomposition), so the necessary and sufficient condition we must check for the non-singularity is:

$$Det(A) \neq 0 \iff \prod_{i=1}^{n} u_{ii} \neq 0 \tag{A.1}$$

where u_{ii} are the diagonal elements of U. Let us prove this statement.

We express A as $A = LU$ using its LU decomposition. Thus:

$$A = LU \implies \mathrm{Det}(A) = \mathrm{Det}(LU) \implies \mathrm{Det}(A) = \mathrm{Det}(L)\mathrm{Det}(U) = \mathrm{Det}(U) = \prod_{i=1}^{n} u_{ii} \tag{A.2}$$

where we have considered that both L and U are triangular matrices, so their determinant is just the product of their diagonal elements. Taking into account that L has only 1 in the diagonal, its determinant is 1, so there is only left the determinant of matrix U. The proof of the fact that Det(AB) = Det(A)·Det(B) for any given two matrices A, B can be found in any elementary linear algebra book, such as [5].

B Hilbert Matrix of dimension 10

```
H = hilbert(10)
print(H)

--> output
[[1.          0.5         0.33333333 0.25        0.2         0.16666667
  0.14285714 0.125       0.11111111 0.1        ]
 [0.5         0.33333333 0.25        0.2         0.16666667 0.14285714
  0.125      0.11111111 0.1         0.09090909]
 [0.33333333 0.25        0.2         0.16666667 0.14285714 0.125
  0.11111111 0.1         0.09090909 0.08333333]
 [0.25        0.2         0.16666667 0.14285714 0.125      0.11111111
  0.1        0.09090909 0.08333333 0.07692308]
 [0.2         0.16666667 0.14285714 0.125      0.11111111 0.1
  0.09090909 0.08333333 0.07692308 0.07142857]
 [0.16666667 0.14285714 0.125      0.11111111 0.1        0.09090909
  0.08333333 0.07692308 0.07142857 0.06666667]
 [0.14285714 0.125      0.11111111 0.1        0.09090909 0.08333333
  0.07692308 0.07142857 0.06666667 0.0625    ]
 [0.125      0.11111111 0.1        0.09090909 0.08333333 0.07692308
  0.07142857 0.06666667 0.0625     0.05882353]
 [0.11111111 0.1        0.09090909 0.08333333 0.07692308 0.07142857
  0.06666667 0.0625     0.05882353 0.05555556]
 [0.1        0.09090909 0.08333333 0.07692308 0.07142857 0.06666667
  0.0625     0.05882353 0.05555556 0.05263158]]
```

Listing 21: Hilbert matrix of dimension 10.

C Codes from the Manual

In this section we add the codes that were already implemented in the Manual [2] as well as the comments we added.

```
def LU(a0, pivotMethod=partialPivoting):
    """ Generate a LU decomposition with partial pivoting

    Paramters:
    ----------
    a0: numpy array, input matrix
    pivotMethod: function

    Returns:
    --------
    a          : numpy matrix, has in the upper triangle the matrix U
                 and below the main diagonal the matrix L
    z          : numpy array, permutation vector which uniquely defines the
    permutation
                 matrix P
                 e.g. [2, 0, 1] --> [[0, 0, 1], [1, 0, 0], [0, 1, 0]]
    det_out    : determinant of the matrix a0
    sing_FLAG : bool, True if a0 is singular, false otherwise
    """
    a = np.copy(a0)
    n = a.shape[0]
    # Initialize the row permutation vector
    permut_vec = np.array([i for i in range(n)])
    for j in range(n-1):
        p, apivot = pivotMethod(a, j)
        if apivot != 0:
            #Now flip row j and p
            for i in range(n):
                a[j, i], a[p, i] = a[p, i], a[j, i]

            #and log this in the permutation vector
            permut_vec[j], permut_vec[p] = permut_vec[p], permut_vec[j]

            #Elimination step
            for i in range(j+1, n):
                a[i, j] = a[i, j]/a[j, j]
                for k in range(j+1, n):
                    a[i, k] = a[i, k] - a[i, j]*a[j, k]


    #find out if matrix is singular
    sing_FLAG = False
    idet = 1
    num_acc = 1.0e-10
    for j in range(n): #the determinant is the product of the diagonal, since we
    are using the triangular version of it
        if np.abs(a[j, j]) < num_acc:
            idet = 0 #if any element of the diagonal is 0, det = 0
    if idet == 0:
        print("lrloes_gb is not applicable.")
        sing_FLAG = True
    return a, permut_vec, sing_FLAG
```

Listing 22: LU funtion.

```
def get_mat_err(mat, pivotMethod=partialPivoting):
    """ Return the error of the matrix inversion algorithm
    calculated as sum(abs((A^{-1})^{-1} - A))

```

```
    Parameters:
    -----------
    mat: numpy array, input matrix A
    pivotMethod: function

    Returns:
    --------
    error: float, max(abs(A * A^{-1} - I))
    diff: float, execution time for the matrix inversion routine
    """

    start_time = time.time()
    inverse = solve_eq(mat, pivotMethod=pivotMethod)
    diff_time = time.time() - start_time
    return np.sum(abs(inverse.dot(mat) - np.identity(mat.shape[0]))), diff_time

def get_bench(fp_type, n_size, pivotMethod=partialPivoting):
    """Benchmark algorithm for a certain floating
    point precision.

    Parameters:
    -----------
    n_size: array of floates, specifies the array sizes on which
            to calculate the matrix error
    fp_type: object of np.dtype, specifies the floating point
             precision (possible are float16, float32, float64)
    pivotMethod: pivoting method, True if pivoting is used

    Returns:
    --------
    bench_inv: numpy array, array with column entries:
               matrix_size, error, execution_time
    """

    bench_inv = np.zeros((len(n_size), 3))
    for idx, el in enumerate(n_size):
        err = get_mat_err(np.array(np.random.random((el, el)), fp_type),
    pivotMethod=pivotMethod)
        bench_inv[idx, 0] = el
        bench_inv[idx, 1:3] = err

    return bench_inv
```

Listing 23: Benchmarking for matrix inversion algorithms.

```
def get_bench_hilbert(fp_type, n_size, pivotMethod=partialPivoting):
    """Benchmark algorithm for a certain floating
    point precision.

    Parameters:
    -----------
    n_size: array of floates, specifies the array sizes on which
            to calculate the matrix error
    fp_type: object of np.dtype, specifies the floating point
             precision (possible are float16, float32, float64)
    pivotMethod: function

    Returns:
    --------
    bench_inv: numpy array, array with column entries:
               matrix_size, error, execution_time
    """

    bench_inv = np.zeros((len(n_size), 3)) #nx3 (n is the quantity of matrices we
    have) matrices that store in the frist column the size of the matrix, in the
```

```
    second the computational time and in the third the error

    for idx, el in enumerate(n_size):
        err = get_mat_err(np.array(hilbert(el), fp_type), pivotMethod=pivotMethod)
        bench_inv[idx, 0] = el
        bench_inv[idx, 1:3] = err

    return bench_inv
```

Listing 24: Hilbert matrices algorithm.