

# Chapter 1

## Problem 1

In the first problem it is required to write a function that computes the LU factorization of a non-singular matrix  $A$  without pivoting. The full code has been written using Python as programming language and it is available at the following link on GitHub [https://github.com/nunziacerrato/Numerical\\_Analysis\\_Optimization](https://github.com/nunziacerrato/Numerical_Analysis_Optimization). In the following it is reported the function `lufact(A)` that can be found in the library named `Project_1.py`

```
1 def lufact(A):
2     r''' This function computes the LU factorization of a non-singular matrix A
        ↳ without pivoting, giving as output the matrices L and U and the growth
        ↳ factor g, here defined as :math:\frac{\max_{ij} (|L|+|U|)_{ij}}{\max_{ij} |A|_{ij}}.
3
4     Paramters:
5     -----
6     A : ndarray
7         input matrix of dimension :math:(N \times N)
8
9     Returns
10    -----
11    L : ndarray
12        Unit lower triangular matrix
13    U : ndarray
14        Upper triangular matrix
15    g : float
16        growth factor
17    '''
18
19    # Compute the dimension of the input square matrix
20    dim = A.shape
21    n = dim[0]
22
23    # Define the chosen precision
24    precision = np.finfo(float).eps
25
26    # Check that the input matrix is a square matrix
27    assert (dim[0] == dim[1]), "The input matrix is not a square matrix"
28
29    # Check if the input matrix is singular
30    if np.abs(np.linalg.det(A)) < precision:
```

```

31     logging.warning("The input matrix is singular")
32     # Check if the hypothesis of the LU factorization theorem hold
33     for k in range(n):
34         if np.abs(np.linalg.det(A[:k+1,:k+1])) < precision:
35             logging.warning(f'The {k}-th principal minor is less than the chosen
36                 ↪ precision')
37     # Create a copy of the input matrix to be modified in order to obtain the
38     ↪ matrices L and U
39     B = np.copy(A)
40     for k in range(0,n-1):
41         for i in range(k+1,n):
42             B_kk = B[k,k]
43             # Check if there is a division by a quantity smaller than the chosen
44             ↪ precision
45             if np.abs(B_kk) < precision:
46                 raise ValueError('Division by a quantity smaller than the chosen
47                     ↪ precision - B_kk = {B_kk}')
48             B[i,k] = B[i,k]/B_kk
49         for j in range(k+1,n):
50             for l in range(k+1,n):
51                 B[l,j] = B[l,j] - B[l,k]*B[k,j]
52     # Extract the matrices L and U from B using, respectively, a strictly lower
53     ↪ triangular mask and an upper triangular mask.
54     L = np.tril(B,k=-1) + np.eye(n) # Add the Id matrix in order for L to be
55     ↪ unit lower triangular
56     U = np.triu(B,k=0)
57     # Compute the growth factor
58     LU_abs = np.abs(L)@ np.abs(U)
59     g = np.amax(LU_abs)/np.amax(np.abs(A))
60     return L, U, g

```

We included tests in this function in order to ensure that the input matrix has the correct characteristics to perform the LU factorization. The rationale used to insert such tests is explained in the following. We added *warnings*, using the `logging` Python library, for all the checks that, if not satisfied, do not necessarily break the LU factorization, so the flow of the code does not interrupt. For what concern most significant errors, we inserted an *assertion* if the input matrix is not square, so that the flow of the code is interrupted, and a *ValueError* if a division by a quantity smaller than the chosen precision occurs. In this last case, if the LU factorization fails, the exception is handled in the main program and this information is stored in a counter (see subsection 1.1), that is saved in an Excel file for all the types of matrices and the dimensions considered. The counter will be updated each time a failure occurs.

In analyzing the results regarding the correctness of the computational LU factorization for the chosen input matrices, it is required to study the trend of the growth factor and the relative backward error with respect to the dimension of the considered matrices. While the growth factor is computed within the function `lufact(A)`, the relative backward error is computed separately. The code that computes this value is reported below.

```

1 def relative_backward_error(A,L,U):
2     r''' This function computes the relative backward error of the LU
        ↪ factorization, defined as :math:\frac{\|A - LU\|}{\|A\|}
        ↪ \frac{\|A - LU\|}{\|A\|}
3
4     Parameters
5     -----
6     A : ndarray
7         Input matrix
8     L : ndarray
9         Unit lower triangular matrix, obtained from the LU factorization of the
        ↪ input matrix A.
10    U : ndarray
11        Upper triangular matrix, obtained from the LU factorization of the
        ↪ input matrix A.
12
13    Returns
14    -----
15    out : float
16        Relative backward error
17    '''
18
19    return np.linalg.norm(A - L @ U, ord=np.inf)/np.linalg.norm(A, ord=np.inf)

```

Note that the symbol @ stands for the operator that performs the matrix multiplication.

Once having constructed these functions, we built the dataset of matrices, on which apply the LU factorization, by creating a function that takes two values as input, namely the number of matrices of each type and their dimension, and gives as output a dictionary containing all the sampled matrices of each type. In particular, we considered random matrices, unitary matrices, Hermitian matrices, positive definite matrices and diagonally dominant matrices. For the first type of matrices, we distinguished matrices whose entries are real and uniformly sampled in the range  $[0, 1)$  and matrices whose entries are independent complex and normally distributed. To sample unitary and Hermitian matrices we considered, instead, ensembles of random matrices using the `tenpy` library in Python. Positive definite matrices are sampled using the `qutip` library in Python, by considering trace-one matrices of the form  $A^*A$ , where  $A^*$  stands for the Hermitian conjugate of the matrix  $A$ , sampled as a matrix whose entries are independent complex and normally distributed. Finally, we define a function to sample diagonally dominant matrices, which is reported below.

In this last case the idea is to generate random matrices whose entries are normally distributed and substitute each diagonal element with a new one obtained by summing the absolute values of all the elements in the corresponding row (including itself). The sign of each diagonal element is chosen at random, by generating  $N$  numbers in the interval  $[0, 1)$  and applying the sign function to these numbers, shifted by 0.5.

```

1 def diagonally_dominant_matrix(N):
2     ''' This function returns a diagonally dominant matrix of dimension
        ↪ :math:(N \times N), whose non-diagonal entries are normally
        ↪ distributed.
3
4     Parameters

```

```

5  -----
6  N : int
7      Dimension of the output matrix
8
9  Returns
10 -----
11 out : ndarray
12     Diagonally dominant matrix
13
14 '''
15 # The following steps are made to decide the sign of the diagonal element
16 → of the output matrix
17 # Obtain N random numbers in [0,1) and apply the sign function to this
18 → values, shifted by 0.5
19 diag_sign = np.random.rand(N)
20 diag_sign = np.sign(diag_sign - 0.5)
21 diag_sign[diag_sign == 0] = 1 # Set to 1 the (very unlikely) values equal
22 → to 0
23
24 # Obtain a matrix of dimension (NxN) whose entries are normally distributed
25 M = np.random.normal(loc=0.0, scale=1.0, size=(N,N))
26 # Substitute all the diagonal elements in this matrix with the sum of the
27 → absolute values of all the elements in the corresponding row (including
28 → itself)
29 for i in range(N):
30     M[i,i] = sum(np.abs(M[i,:])) * diag_sign[i]
31
32 return M

```

We have then defined a function that creates the dataset containing all the types of matrices listed before. Since that part of the code is trivial, we have decided to not report it here for brevity (it can be found [here](#) in the online repository on GitHub). We do a brief description of how we organize such function. First and foremost, the function `create_dataset` takes as input the number of matrices of each type and their relative dimension and gives as output a dictionary whose keys represent the different types of matrices considered and whose values are 3-dimensional arrays, where the first index cycles on the number of matrices considered. The output matrices are chosen to be nonsingular. We decided to use dictionaries to define the result of this function in order to have a better control the flow of the code in the main program and ensure an easily readability of the input data. Moreover, we fixed the seed in order to have reproducibility of the results (`tenpy` and `qutip` libraries work using `numpy` in the background, so this choice for the seed holds for all the sampled matrices).

## 1.1 Main program

In the main program, after having imported all the necessary Python libraries, we created the dataset and computed the LU factorization for all the types of matrices considered, while also taking into account different dimensions. All the data are saved in Excel files. We also created a DataFrame to store all the failures of the algorithm, where column names represent the different types of matrices considered, row indices represent the progressive dimension of the matrices, while elements of the DataFrame represent the total number of failures of the LU factorization for a certain matrix type of a given dimension.

```

1  # Define global parameters
2  num_matr = 500
3  dim_matr_max = 50
4  common_path =
   ↪  "C:\\Users\\cerra\\Documents\\GitHub\\Numerical_Analysis_Optimization\\Project_1"
5
6  keys = create_dataset(1,2).keys()
7
8  # Define a DataFrame to store all the failures of the LU factorization
   ↪  divided by matrix types.
9  df_fails = pd.DataFrame(0, columns = keys, index = range(2,dim_matr_max+1))
10
11 # Cycle on the different dimensions considered
12 for dim_matr in range(2,dim_matr_max+1):
13
14     # Create the dataset
15     dataset = create_dataset(num_matr, dim_matr)
16
17     # Create DataFrames in which the growth factor and the relative backward
   ↪  error are stored
18     df_g = pd.DataFrame(columns = keys)
19     df_rel_back_err = pd.DataFrame(columns = keys)
20
21     # Cycle on the different types of matrices considered
22     for matrix_type in keys:
23
24         # Cycle on the number of matrices of each type
25         for i in range(num_matr):
26             # Select the matrix and compute the LU factorization, the growth factor
   ↪  and the relative backward error
27             A = dataset[matrix_type][i,:,:]
28             try:
29                 L, U, df_g.at[i,matrix_type] = lufact(A)
30                 df_rel_back_err.at[i,matrix_type] = relative_backward_error(A, L, U)
31             except ValueError:
32                 df_fails.at[dim_matr,matrix_type] = df_fails.at[dim_matr,matrix_type] +
   ↪  1
33
34     # Save the growth factor and the relative backward error in Excel files
35     writer = pd.ExcelWriter(f'{common_path}\\Data\\'
36 f'Statistics_for_{num_matr}_matrices_of_dim_{dim_matr}.xlsx')
37     df_g.to_excel(writer, 'growth_factor', index = False)
38     df_rel_back_err.to_excel(writer, 'rel_back_err', index = False)
39     writer.save()
40
41     # Save the failures of the LU factorization in an Excel file
42     writer = pd.ExcelWriter(f'{common_path}\\Data\\'
43 f'Failures_LUfact_for_{num_matr}_matrices.xlsx')
44     df_fails.to_excel(writer, 'Fails', index = False)
45     writer.save()

```

Choosing this approach, once having created the dataset and computed the quantity of in-

terest, we do not have to run the algorithm again, as all the data are now stored. Different scripts are dedicated to compute the minimum and the maximum value of the growth factor and of the relative backward error, as well as the mean value and the standard deviation of such quantities. For the sake of simplicity, we do not report here such part of the code, as it is of no particular relevance. However, it can be found [here](#) in the online repository on GitHub.

## Results

We considered 500 samplings for all the types of non-singular, square matrices investigated, whose dimension varies from 2 to 50.

**Random Matrices** We generated two different types of random matrices, namely real matrices with entries sampled uniformly in the interval  $[0, 1)$ , which we simply call random matrices, and complex matrices with entries of the form  $a+ib$ , where  $a$  and  $b$  are independently sampled from the normal distribution. The latter are called *Ginibre matrices*. We made this choice to construct a dataset as representative as possible, trying to take into consideration different types of distributions for the entries of the matrices. We report in the following the plot of the characteristic values of the growth factor  $\gamma$  and the relative backward error  $\delta$  as a function of the dimension  $N$  of the matrix.

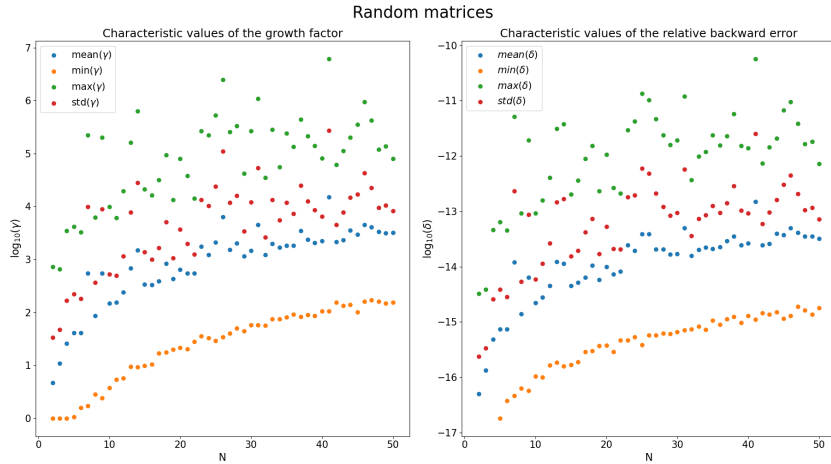


Figure 1.1: Scatterplot of the characteristic values of  $\gamma$  and  $\delta$  as a function of  $N$  for real random matrices. Logarithmic scale on the ordinate axis.

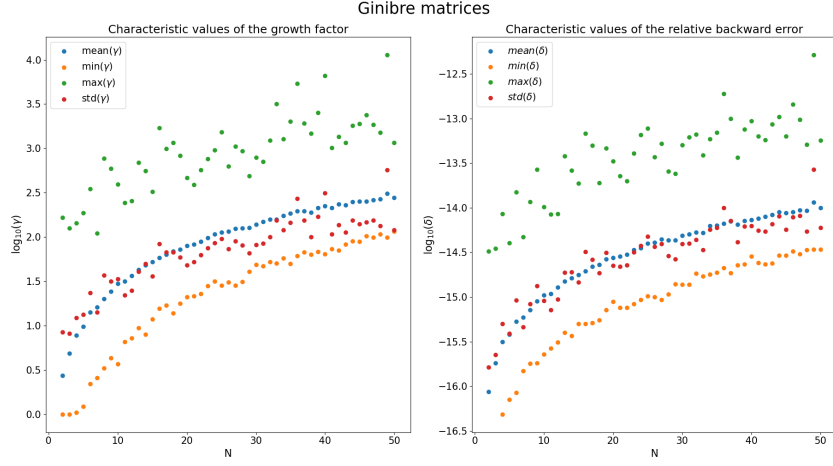


Figure 1.2: Scatterplot of the characteristic values of  $\gamma$  and  $\delta$  as a function of  $N$  for Ginibre matrices. Logarithmic scale on the ordinate axis.

As can be seen from these plots, the characteristic values of the two considered quantities increase with the dimension of the input matrix for both classes of matrices taken into account. This might be due to the fact that, when the dimension of the input matrix increases, the algorithm that performs the LU factorization naturally requires more steps, resulting in an unavoidable propagation of numerical errors. It is interesting to also consider, for a fixed dimension, the distributions of the growth factor and the relative backward error for both cases. We have chosen to report here the histograms obtained for  $N = 25$ , together with the corresponding boxplots, where the outliers have been removed for clarity of visualization. We have chosen this point as an intermediate representative dimension; similar considerations also hold in the other cases.

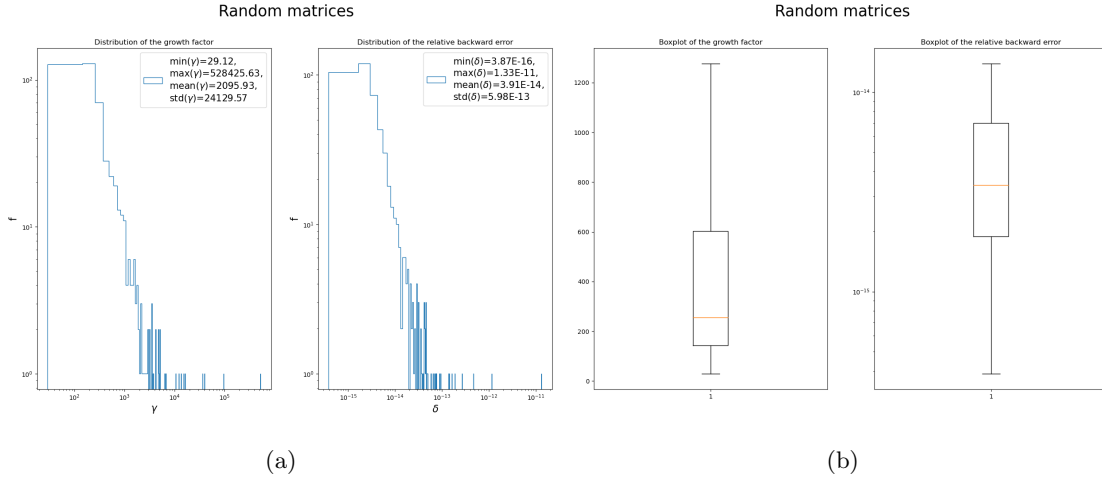


Figure 1.3: Histograms (panel (a)) and boxplots (panel (b)) of  $\gamma$  and  $\delta$  considering real random matrices of dimension  $25 \times 25$ . Logarithmic scale is reported on both axis of the histograms as well as on the ordinate axes of the backward error boxplot.

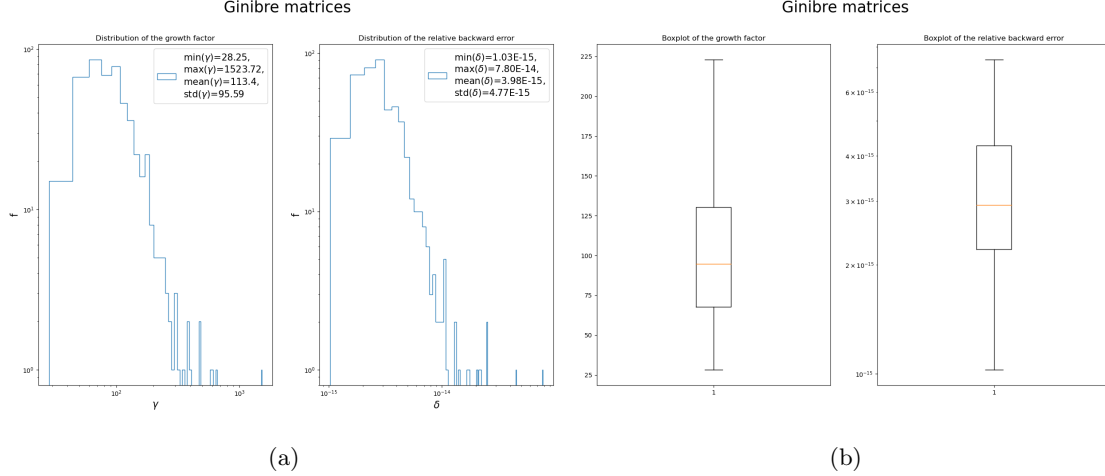


Figure 1.4: Histograms (panel (a)) and boxplots (panel (b)) of  $\gamma$  and  $\delta$  considering Ginibre matrices of dimension  $25 \times 25$ . Logarithmic scale is reported on both axis of the histograms as well as on the ordinate axes of the backward error boxplot.

It can be observed that both the growth factor and the relative backward error have a wider range of variation, if one also consider outliers, in the case of real random matrices, while these ranges are smaller in the case of Ginibre matrices. This is, in general, an indication of the fact that both values strongly depend on the input matrix, even if more careful considerations can be made to distinguish the two cases. In fact, taking into account, for example, the relative backward error and considering also the outliers, one can note that for real random matrices, whose entries are uniformly distributed in  $[0, 1)$ , this quantity ranges approximatively from  $10^{-16}$  to  $10^{-11}$ . For what concern the Ginibre matrices, the same quantity ranges approximatively from  $10^{-15}$  to  $10^{-13}$ . It is worth noting that the presence of outliers is more marked in the first case. This difference in the variation range might be due to the fact that when considering the normal distribution to sample the entries of a random matrix, it is more likely to have matrix elements that are centred in the neighbourhood of the mean value, resulting in an ensemble of more homogeneous matrices. This, as a consequence, might reflect in the obtained values of relative backward errors, when computing the LU factorization, that may be similar and with a smaller range of variation. In the other case, when considering the uniform distribution to sample the entries of a random matrix, the resulting ensemble of matrices may be less homogeneous when compared to the previous one, and this might result in a wider range of variation of the quantity under examination. However, even if in both cases the relative backward errors grow with the dimension of the matrix, their values are small enough and, at least for the mean value, not that far from the machine epsilon, that is of order  $10^{-16}$ , so the algorithm can be considered backward stable for these types of matrices.

**Unitary matrices** We considered in our dataset the ensemble of unitary matrices, namely matrices  $U$  such that  $U^* = U^{-1}$ , where  $*$  stands for the conjugate transpose. These matrices have been sampled using the `tenpy` Python library and they are also called *CUE matrices*, as they are sampled from the circular ensemble. As in the previous case, we report in the following the scatterplot of the growth factor  $\gamma$  and the relative backward error  $\delta$  as a function of the dimension  $N$  of the matrix.



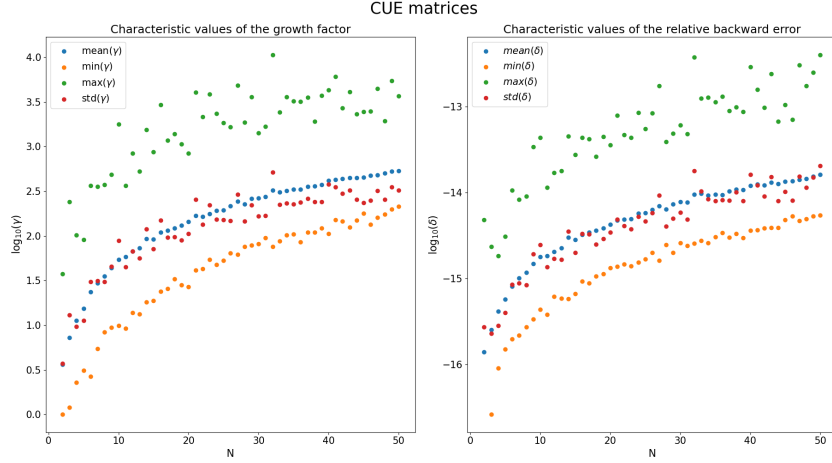


Figure 1.5: Scatterplot of the characteristic values of  $\gamma$  and  $\delta$  as a function of  $N$  for unitary matrices. Logarithmic scale on the ordinate axis.

As can be seen from the values obtained for  $\gamma$  and  $\delta$ , the LU factorization algorithm can be considered backward stable for unitary matrices. Note that the range of variation of both these quantities is comparable with the one obtained in the case of Ginibre matrices.

**Hermitian matrices** The next class of matrices that we considered is that of Hermitian matrices, namely matrices  $H$  such that  $H = H^*$ . In order to sample these matrices we used the `tenpy` Python library. These matrices are also called *GUE matrices* as they are sampled from the Gaussian unitary ensemble.

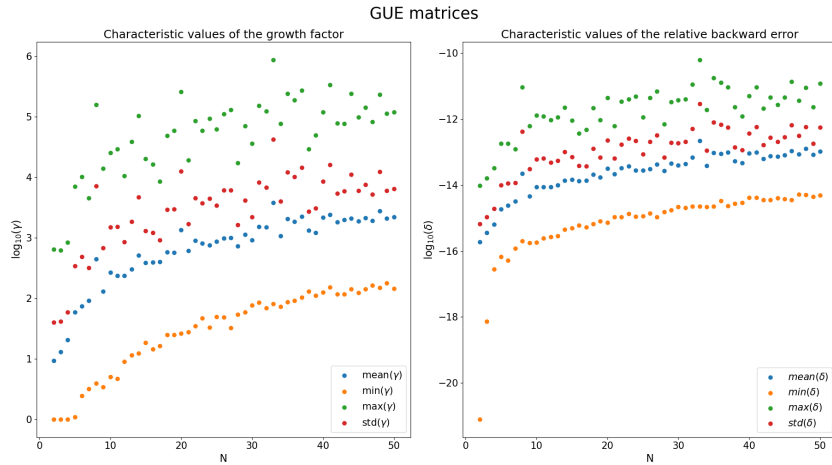


Figure 1.6: Scatterplot of the characteristic values of  $\gamma$  and  $\delta$  as a function of  $N$  for Hermitian matrices. Logarithmic scale on the ordinate axis.

A common characteristic of all these scatterplots is that the range of variation of both the growth factor and the relative backward error is quite large, approximatively of two or three order of magnitude. The choice input matrix has, therefore, an impact in determining these values. However, as in the previous cases, these data suggest that the algorithm is backward stable for this class of matrices.

**Positive definite matrices** The next class of matrices that we considered is that of positive definite matrices. Positive definite matrices  $W$  can be obtained by considering matrices of the form  $W = A^*A$ , where  $*$  stands for the conjugate transpose, discarding singular matrices. If the matrix  $A$  is sampled from the Ginibre ensemble, that is it is a complex matrix whose

entries are independent normally distributed, the matrix  $W$  is called *Wishart matrix*. In order to sample these matrices we used the `qutip` Python library, considering, in particular, unit trace matrices. Note that matrices of this kind are also Hermitian. We expect the LU factorization for these matrices to be backward stable.

We report in the following the scatterplot of the growth factor  $\gamma$  and the relative backward error  $\delta$  as a function of the dimension  $N$  of the input matrix.

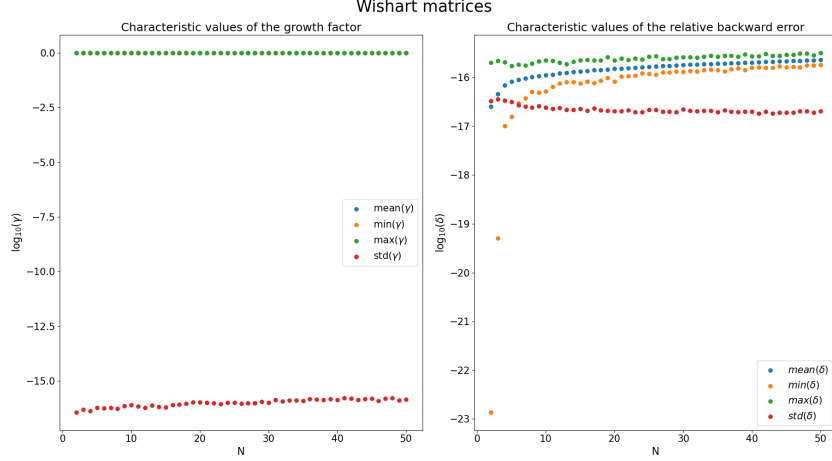


Figure 1.7: Scatterplot of the characteristic values of  $\gamma$  and  $\delta$  as a function of  $N$  for positive definite matrices. Logarithmic scale on the ordinate axis.

It turns out that this case is quite different from the previous ones. In fact, it can be seen that the minimum, maximum and mean value of the growth factor are all equal to one, while the standard deviation is of the order of the machine epsilon. Moreover, the relative backward error is significantly small and it is of the order of the machine epsilon. It is interesting to note that, in this latter case, the standard deviation is smaller if compared to the other values, suggesting that the distribution of the relative backward error is peaked in correspondence with the mean value. This means that, differently from the other cases, whatever the input matrix, the values obtained for the relative backward error will be approximately the same or, in other words, they will be included in a very small range. These considerations imply the backward stability of the LU factorization, as expected, for this class of matrices.

**Diagonally dominant matrices** Another class of matrices that we considered in our dataset is given by the diagonally dominant matrices. These are matrices whose diagonal entries are, in absolute value, greater or equal to the sum of the absolute values of the other elements in the corresponding row, and that the inequality is strict for at least a row. We have described how we sampled this matrices in the description of the function at the beginning of the chapter. In the following we report the scatterplot of the growth factor  $\gamma$  and the relative backward error  $\delta$  as a function of the dimension  $N$  of the matrix.

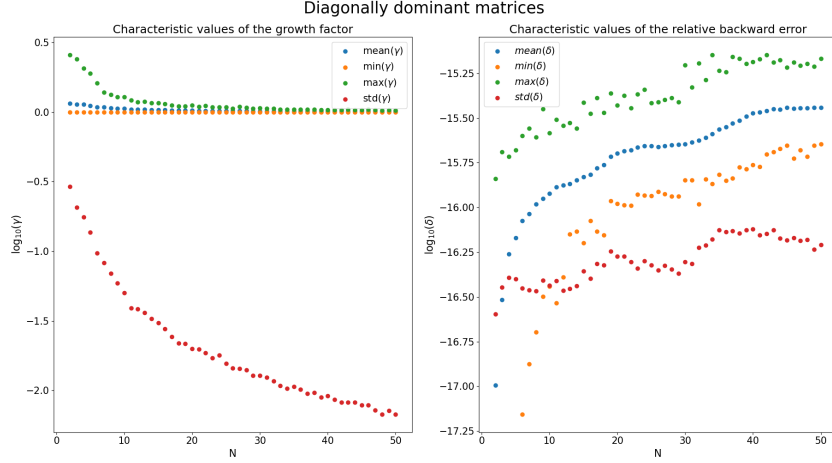


Figure 1.8: Scatterplot of the characteristic values of  $\gamma$  and  $\delta$  as a function of  $N$  for diagonally dominant matrices. Logarithmic scale on the ordinate axis.

In this case, considerations similar to the case of positive definite matrices can be made. In fact, it can be observed that the growth factor is of order  $\mathcal{O}(1)$ , while the relative backward error is of the order of the machine epsilon, with the standard deviation that suggests, as in the previous case, that the distribution of this quantity, for a fixed dimension of the input matrix, is peaked at the mean value. Therefore, the value obtained for this quantity does not strongly depend on the input matrix. It can be concluded that the LU factorization is backward stable for this class of matrices.

**Hilbert matrices** In addition to the functions considered in the dataset, we also analyzed the trend of the growth factor and the relative backward error for the Hilbert matrices  $H_N$ , as a function of their dimension. In this case, it emerges that, when considering a precision given by the machine epsilon  $\varepsilon$ , the algorithm that performs the LU factorization breaks for  $N \geq 21$ . In other words, the growth factor is identically equal to one and the relative backward error is of the order of  $\varepsilon$  for  $N < 21$ , while, as  $N$  increases, a division by a quantity smaller than the machine epsilon occurs, causing the algorithm to break. This means that the problem of solving a linear system  $H_N \mathbf{x} = \mathbf{b}$  is ill-conditioned for  $N \geq 21$  as the matrix becomes numerically singular, that is to say its condition number increases. In fact, we know from theory that the condition number of the Hilbert matrix increases exponentially with the dimension of the matrix.

## Conclusions

Given the classes of matrices considered in the dataset, we found that the LU factorization is backward stable in all the cases considered, with differences that emerge when comparing the last two types of matrices, namely positive definite and diagonally dominant matrices, to the other ones. In fact, in the last two cases, the values obtained for the relative backward error do not depend on the input matrix, differently from the other cases, where this happens. Moreover, we observed that the LU factorization algorithm never failed, as the DataFrame where we stored all the failures of the algorithm, divided by type of matrix considered and its relative dimension, is identically zero. Different considerations hold for the Hilbert matrices, where the LU factorization algorithm breaks for dimensions of the input matrix greater or equal to 21 (given the chosen precision), as a consequence of the increasing condition number. In the following we report a final plot where we compare the boxplots of all the types of matrices taken into account when considering a fixed dimension of the input matrix, that is  $N = 25$ . For greater clarity of visualization, we discard outliers and consider, in this way,

the most likely range of possible values.

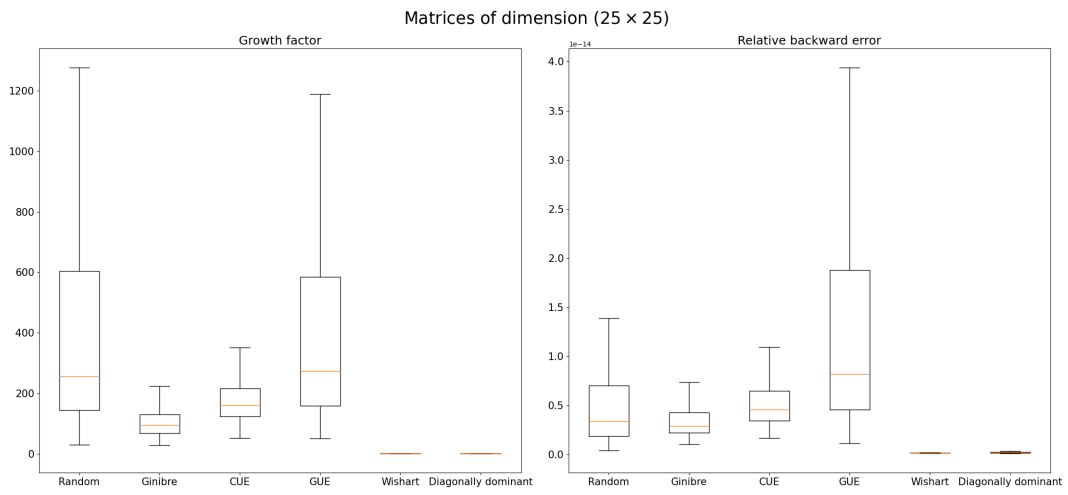


Figure 1.9: Boxplot of  $\gamma$  and  $\delta$  for matrices of dimension  $25 \times 25$ .