Numerical Linear Algebra Homework Project 1: Solution of Linear Systems

Catalano Giuseppe, Cerrato Nunzia

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. In the following it is reported the function `lufact(A)` that can be found in the library named Project_1.py

```
def lufact(A):
 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}}`.

 Paramters:
 ----------
 A : ndarray
     input matrix of dimension :math:`(n\times n)`

 Returns
 -------
 L : ndarray
     Unit lower triagular matrix
 U : ndarray
     Upper triangular matrix
 g : float
     growth factor
 '''

 # Compute the dimension of the input square matrix
 dim = A.shape
 n = dim[0]

 # Define the chosen precision
 precision = np.finfo(float).eps

 # Check that the input matrix is a square matrix
 assert (dim[0] == dim[1]), "The input matrix is not a square matrix"

 # Check if the input matrix is singular
 if np.abs(np.linalg.det(A)) < precision:
    logging.warning("The input matrix is singular")
```

```
    # Check if the hypothesis of the LU factorization theorem hold
    for k in range(n):
        if np.abs(np.linalg.det(A[:k+1,:k+1])) < precision:
            logging.warning(f'The {k}-th principal minor is less than the chosen
            ↪ precision')

    # Create a copy of the input matrix to be modified in order to obatin the
    ↪ matrices L and U
    B = np.copy(A)
    for k in range(0,n-1):
      for i in range(k+1,n):
          B_kk = B[k,k]
          # Check if there is a division by a quantity smaller than the chosen
          ↪ precision
          if np.abs(B_kk) < precision:
              raise ValueError('Division by a quantity smaller than the chosen
              ↪ precision - B_kk = {B_kk}')
          B[i,k] = B[i,k]/B_kk
      for j in range(k+1,n):
          for l in range(k+1,n):
              B[l,j] = B[l,j] - B[l,k]*B[k,j]

    # Extract the matrices L and U from B using, resepctively, a strictly lower
    ↪ triangular mask and an upper triangular mask.
    L = np.tril(B,k=-1) + np.eye(n) # Add the Id matrix in order for L to be
    ↪ unit lower triangular
    U = np.triu(B,k=0)

    # Compute the growth factor
    LU_abs = np.abs(L)@ np.abs(U)
    g = np.amax(LU_abs)/np.amax(np.abs(A))

    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.

```
def relative_backward_error(A,L,U):
```

```
    r''' This function computes the relative backward error of the LU
    ↪  factorization, defined as :math:`\frac{\lVert A -LU
    ↪  \rVert_{\infty}}{\lVert A \rVert_{\infty}}`

    Parameters
    ----------
    A : ndarray
        Input matrix
    L : ndarray
        Unit lower triangular matrix, obtained from the LU factorization of the
        ↪  input matrix A.
    U : ndarray
        Upper triangular matrix, obtained from the LU factorization of the
        ↪  input matrix A.

    Returns
    -------
    out : float
          Relative backward error
    '''

    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 fist 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.

```
def diagonally_dominant_matrix(n):
    ''' This function returns a diagonally dominant matrix of dimension
    ↪  :math:`(n\times n)`, whose non-diagonal entries are normally
    ↪  distributed.

    Parameters
    ----------
```

```
    n : int
        Dimension of the output matrix

    Returns
    -------
    out : ndarray
          Diagonally dominant matrix

    '''
    # The following steps are made to decide the sign of the diagonal element of the output matrix
    # Obtain n random numbers in [0,1) and apply the sign function to this values, shifted by 0.5
    diag_sign = np.random.rand(n)
    diag_sign = np.sign(diag_sign - 0.5)
    diag_sign[diag_sign == 0] = 1 # Set to 1 the (very unlikely) values equal to 0

    # Obtain a matrix of dimension (n × n) whose entries are normally distributed
    M = np.random.normal(loc=0.0, scale=1.0, size=(n,n))
    # Substitute all the diagonal elements in this matrix with the sum of the absolute values of all the elements in the corresponding row (including itself)
    for i in range(n):
       M[i,i] = sum(np.abs(M[i,:])) * diag_sign[i]

    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.

```
# Define global parameters
num_matr = 500
dim_matr_max = 50
common_path = "Project_1"

keys = create_dataset(1,2).keys()

# Define a DataFrame to store all the failures of the LU factorization
↪  divided by matrx types.
df_fails = pd.DataFrame(0, columns = keys, index = range(2,dim_matr_max+1))

# Cycle on the different dimensions considered
for dim_matr in range(2,dim_matr_max+1):

 # Create the dataset
 dataset = create_dataset(num_matr, dim_matr)

 # Create DataFrames in which the growth factor and the relative backward
 ↪  error are stored
 df_g = pd.DataFrame(columns = keys)
 df_rel_back_err = pd.DataFrame(columns = keys)

 # Cycle on the different types of matrices considered
 for matrix_type in keys:

  # Cycle on the number of matrices of each type
  for i in range(num_matr):
   # Select the matrix and compute the LU factorization, the growth factor
   ↪  and the relative backward error
   A = dataset[matrix_type][i,:,:]
   try:
    L, U, df_g.at[i,matrix_type] = lufact(A)
    df_rel_back_err.at[i,matrix_type] = relative_backward_error(A, L, U)
   except ValueError:
    df_fails.at[dim_matr,matrix_type] = df_fails.at[dim_matr,matrix_type] +
    ↪  1

 # Save the growth factor and the relative backward error in Excel files
 writer = pd.ExcelWriter(f'{common_path}\\Data\\'
 f'Statistics_for_{num_matr}_matrices_of_dim_{dim_matr}.xlsx')
 df_g.to_excel(writer, 'growth_factor', index = False)
 df_rel_back_err.to_excel(writer, 'rel_back_err', index = False)
 writer.save()

# Save the failues of the LU factorization in an Excel file
writer = pd.ExcelWriter(f'{common_path}\\Data\\'
f'Failures_LUfact_for_{num_matr}_matrices.xlsx')
df_fails.to_excel(writer, 'Fails', index = False)
writer.save()
```

Choosing this approach, once having created the dataset and computed the quantity of interest, 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.

1.2 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 γ and the relative backward error δ as a function of the dimension N of the matrix.

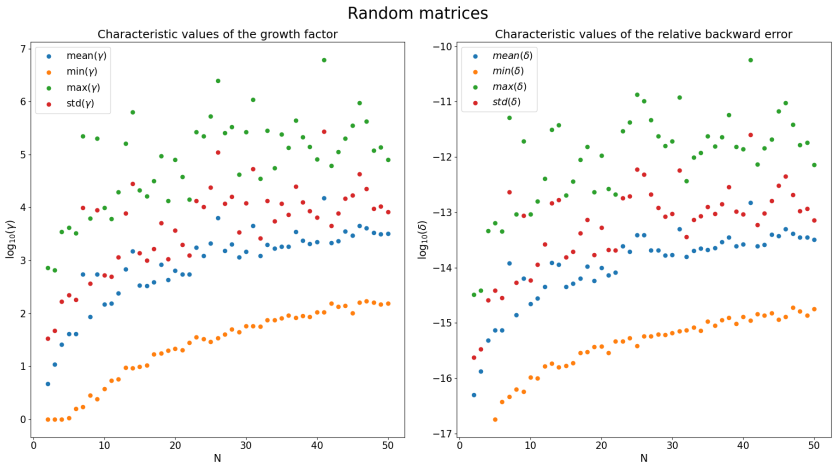


Figure 1: Scatterplot of the characteristic values of γ and δ as a function of N for real random matrices. Logarithmic scale on the ordinate axis.

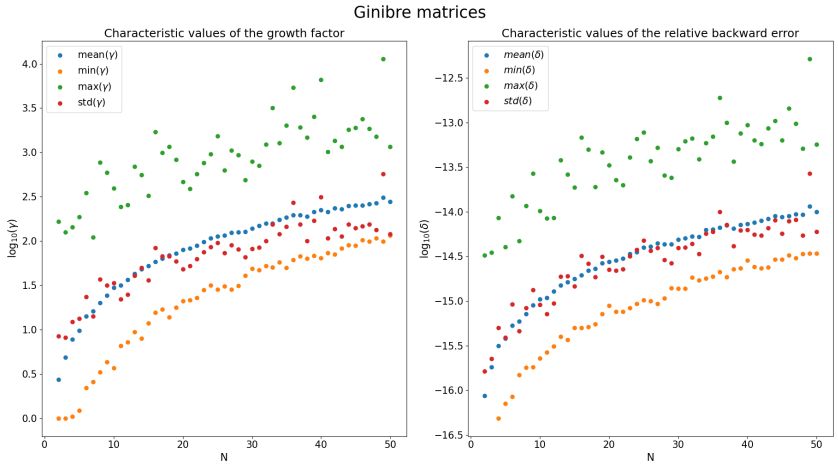
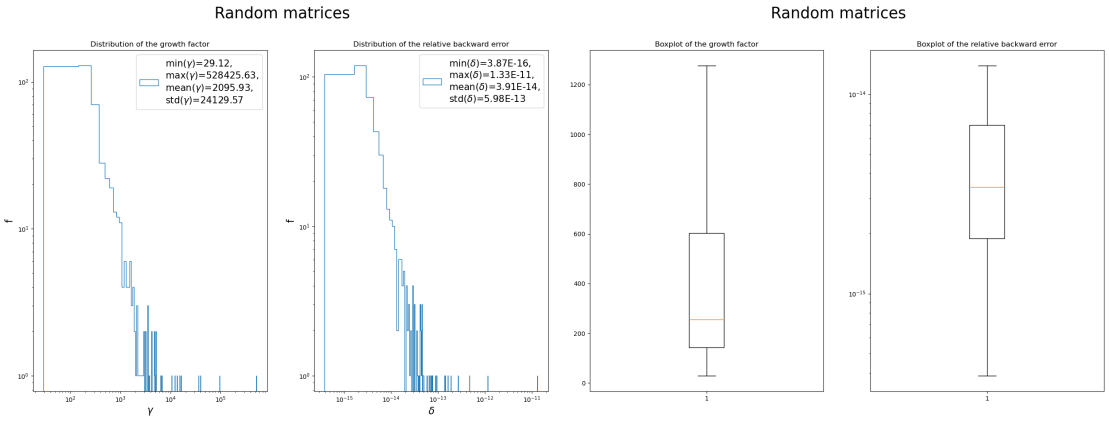


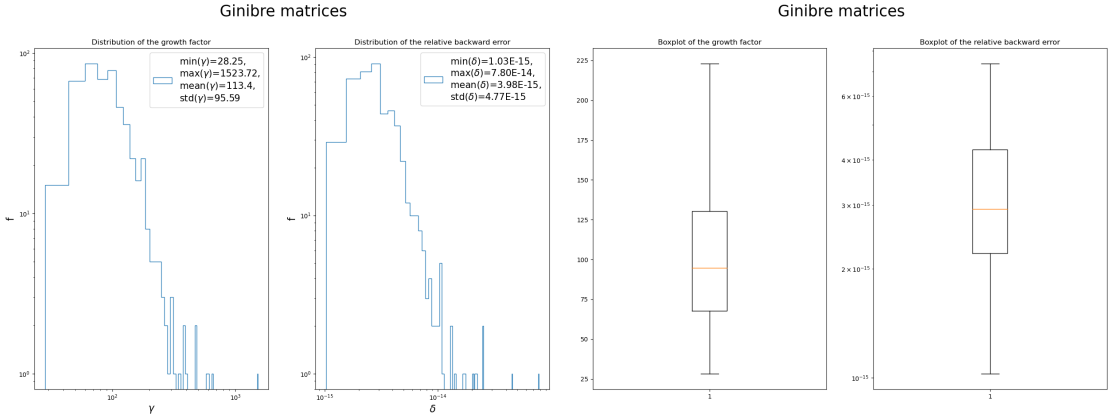
Figure 2: Scatterplot of the characteristic values of γ and δ as a function of N for Ginibre matrices. Logarithmic scale on the ordinate axis.

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.



(a) (b)

Figure 3: Histograms (panel (a)) and boxplots (panel (b)) of γ and δ considering real random matrices of dimension 25×25. Logarithmic scale is reported on both axis of the histograms as well as on the ordinate axes of the backward error boxplot.



(a) (b)

Figure 4: Histograms (panel (a)) and boxplots (panel (b)) of γ and δ considering Ginibre matrices of dimension 25×25. 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 rage 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 out 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 γ and the relative backward error δ as a function of the dimension N of the matrix.

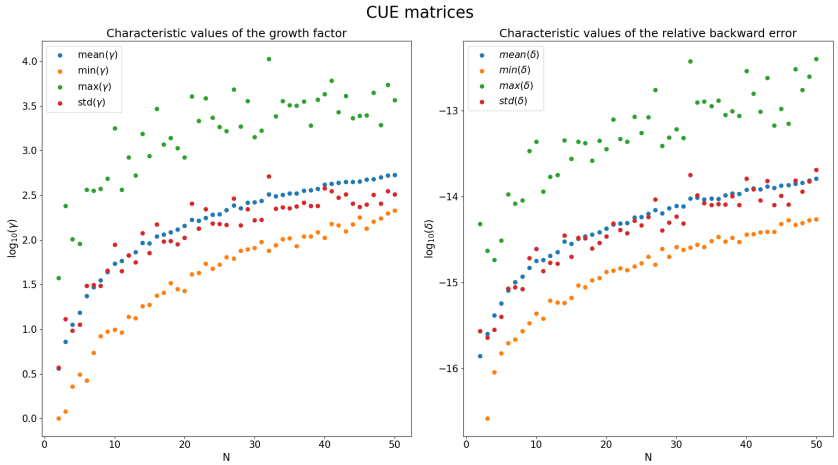


Figure 5: Scatterplot of the characteristic values of γ and δ as a function of N for unitary matrices. Logarithmic scale on the ordinate axis.

As can be seen from the values obtained for γ and δ, 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.

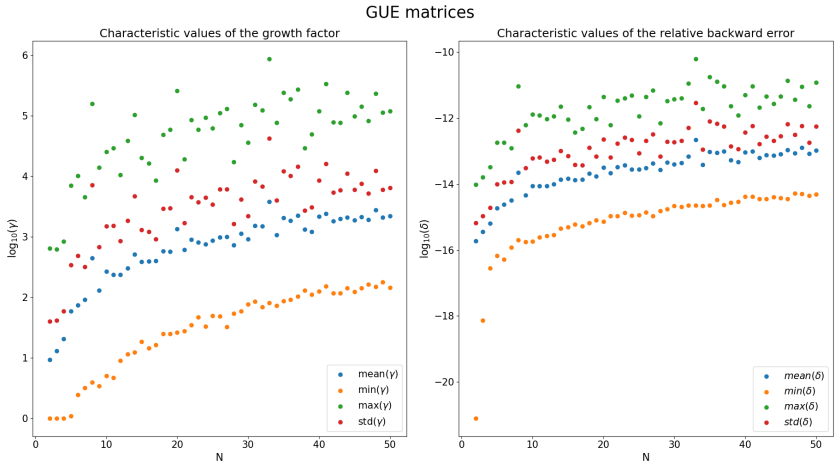


Figure 6: Scatterplot of the characteristic values of γ and δ as a function of N for Hermitian matrices. Logarithmic scale on the ordinate axis.

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 γ and the relative backward error δ as a function of the dimension N of the input matrix.

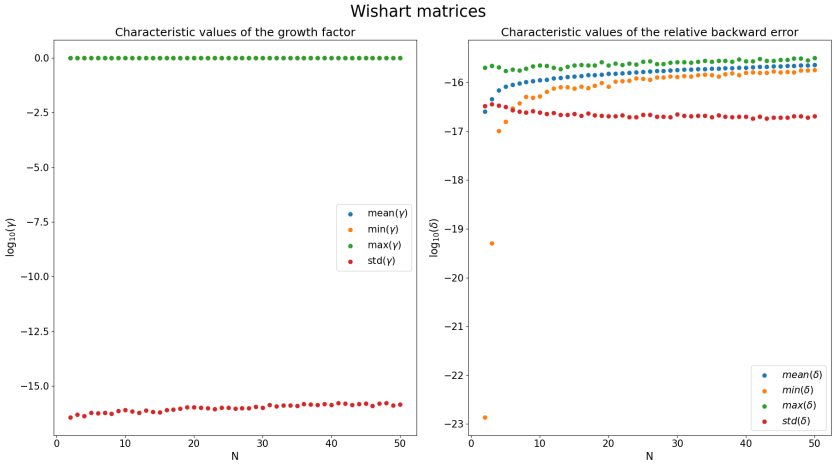


Figure 7: Scatterplot of the characteristic values of γ and δ as a function of N for positive definite matrices. Logarithmic scale on the ordinate axis.

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 γ and the relative backward error δ as a function of the dimension N of the matrix.

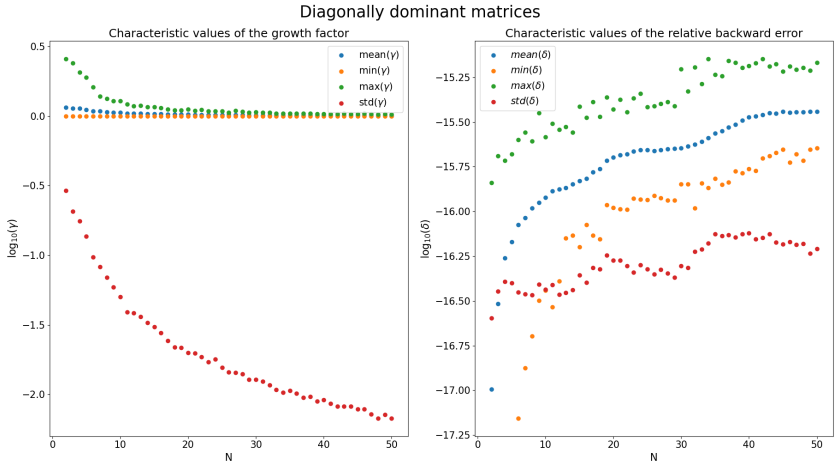


Figure 8: Scatterplot of the characteristic values of γ and δ as a function of N for diagonally dominant matrices. Logarithmic scale on the ordinate axis.

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 ε, 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 ε 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.

1.3 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.

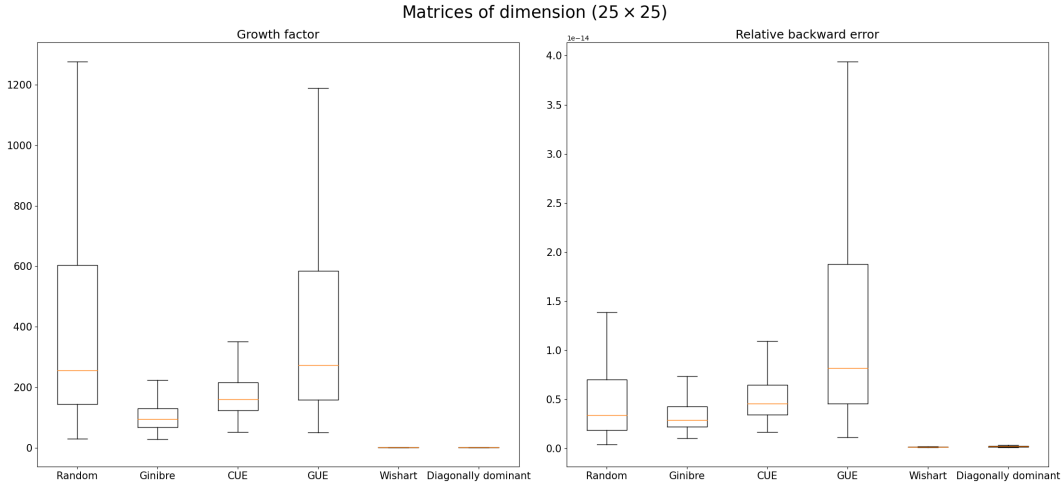


Figure 9: Boxplot of γ and δ for matrices of dimension 25×25.

2 Problem 2

Consider the $n \times n$ Wilkinson matrix

$$W_n = \begin{bmatrix} 1 & 0 & 0 & \cdots & 1 \\ -1 & 1 & 0 & \cdots & 1 \\ -1 & -1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & -1 & 1 \end{bmatrix} \tag{1}$$

(1) We are interested to compute (by hand) the LU factorization of W_5, therefore to compute two $n \times n$ matrices, L_5 and U_5, that are, respectively, a unit lower triangular matrix and an upper triangular matrix that satisfy the identity $W_5 = L_5 U_5$. We start writing the expression of W_5:

$$W_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}, \tag{2}$$

now we compute $\mathbf{m}_1$:

$$\mathbf{m}_1 = \begin{bmatrix} 0 \\ w_{21}/w_{11} \\ w_{31}/w_{11} \\ w_{41}/w_{11} \\ w_{51}/w_{11} \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}. \tag{3}$$

Defining $\mathbf{e}_i$ as the vectors with 1 in the $i-th$ element and 0 otherwise, we can compute:

$$M_1 = \mathcal{I}_5 - \mathbf{m}_1 \mathbf{e}_1^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4}$$

Using the expression of M_1, we can compute $W_5^{(1)}$:

$$W_5^{(1)} = M_1 W_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & -1 & 1 & 0 & 2 \\ 0 & -1 & -1 & 1 & 2 \\ 0 & -1 & -1 & -1 & 2 \end{bmatrix}. \tag{5}$$

The second iteration proceeds in a similar way:

$$\mathbf{m}_2 = \begin{bmatrix} 0 \\ 0 \\ w_{32}^{(1)}/w_{22}^{(1)} \\ w_{42}^{(1)}/w_{22}^{(1)} \\ w_{52}^{(1)}/w_{22}^{(1)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \\ -1 \\ -1 \end{bmatrix}, \tag{6}$$

$$M_2 = \mathcal{I}_5 - \mathbf{m}_2 \mathbf{e}_2^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}. \tag{7}$$

Using the expression of M_2, we can compute $W_5^{(2)}$:

$$W_5^{(2)} = M_2 W_5^{(1)} = M_2 M_1 W_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & -1 & 1 & 4 \\ 0 & 0 & -1 & -1 & 4 \end{bmatrix}. \tag{8}$$

Now we start the third iteration:

$$\mathbf{m}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ w_{43}^{(2)}/w_{33}^{(2)} \\ w_{53}^{(2)}/w_{33}^{(2)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ -1 \end{bmatrix}, \tag{9}$$

$$M_3 = \mathcal{I}_5 - \mathbf{m}_3 \mathbf{e}_3^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}. \tag{10}$$

Using the expression of M_3, we can compute $W_5^{(3)}$:

$$W_5^{(3)} = M_3 W_5^{(2)} = M_3 M_2 M_1 W_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & -1 & 8 \end{bmatrix}. \tag{11}$$

Similarly, we can perform the last iteration:

$$\mathbf{m}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ w_{54}^{(2)}/w_{44}^{(2)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix}, \tag{12}$$

$$M_4 = \mathcal{I}_5 - \mathbf{m}_4 \mathbf{e}_4^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}. \tag{13}$$

Using the expression of M_4, we can compute $W_5^{(4)}$:

$$W_5^{(4)} = M_4 W_5^{(3)} = M_4 M_3 M_2 M_1 W_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & 0 & 16 \end{bmatrix} = U_5. \tag{14}$$

Since this last matrix is upper triangular, we call it U_5 and define $L_5^{-1} \coloneqq M_4 M_3 M_2 M_1$, such that $L_5^{-1} W_5 = U_5$. We can get L_5 from $L_5 = M_1^{-1} M_2^{-1} M_3^{-1} M_4^{-1}$ and knowing that $M_i^{-1} = \mathcal{I}_5 + \mathbf{m}_i \mathbf{e}_i^T$:

$$L_5 = M_1^{-1} M_2^{-1} M_3^{-1} M_4^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}. \tag{15}$$

(2) It is possible to guess the LU factorization of $W_n = L_n U_n$:

$$L_n = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ -1 & 1 & \ddots & & \vdots \\ \vdots & \ddots & 1 & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ -1 & \cdots & \cdots & -1 & 1 \end{bmatrix}, \quad U_n = \begin{bmatrix} 1 & 0 & \cdots & 0 & 2^0 \\ 0 & 1 & \ddots & \vdots & 2^1 \\ \vdots & \ddots & \ddots & 0 & 2^2 \\ \vdots & & \ddots & 1 & \vdots \\ 0 & \cdots & \cdots & 0 & 2^{n-1} \end{bmatrix} \tag{16}$$

(3) In the following, we report the function that generates the $n \times n$ Wilkinson matrix.

```
def wilkin(n):
 r''' This function computes the Wilkinson matrix of dimension :math:`(n
  ↪  \times n)`.

 Parameters
 ----------
 n : int
     Dimension of the Wilkinson matrix

 Returns
 --------
```

```
    W : ndarray
        Wilkinson matrix
    '''
    W = np.tril(-np.ones((n,n)),k=-1) + np.eye(n)
    W[:,n-1] = 1
    return W
```

(4-5-6) In the following, we report the code that performs the numerical experiment for each $n = 2, \dots, 60$.

```
def check_when_lufact_W_fails(n_max = 60, treshold = np.finfo(float).eps):
    r''' This function checks the failures of GEPP for a Wilkinson matrix W_n
     ↪  with dimension less or equal than n_max. We define a failure as the
     ↪  case in which the error between the solution found by the algorithm and
     ↪  the expected solution is higher than a chosen treshold. The default
     ↪  treshold is set equal to the machine epsilon. When an error is found, a
     ↪  warning message is printed. The function returns the list of the
     ↪  dimensions less then or equal to n_max for which the GEPP algorithm
     ↪  fails.

    Parameters
    ----------
    n_max : int
            Maximum size of the Wilkinson matrix

    Returns
    --------
    fails : list
            List of dimensions for which the algorithm fails
    '''

    fails = []

    # Cycle on the dimension of the input
    for n in range(2,n_max+1):
        W = wilkin(n)
        logging.debug(f'W = {W}')

        # Define the vector b
        vect = np.ones(n)
        b = W @ vect

        # Solve the system with GEPP
        x = np.linalg.solve(W,b)

        # Compute the error in 1-norm between the computed solution and the
        ↪  exact solution, and print a warning message when the error exceedes
        ↪  the chosen precision
        error = sum(np.abs(x - vect))
        if error <= treshold:
          logging.info(f'n = {n}, ||x - e||_1 = {error}')
        elif error > treshold:
```

```
        logging.warning(f'n = {n}, ||x - e||_1 = {error}')
        fails.append(n)
        # Cycle on the elements of the computed solution and print a warning
        ↪  message with the wrong elements of the solution.
        for i in range(n):
            if abs(x[i] - 1) > treshold:
                logging.warning(f'x[{i}] = {x[i]}')
return fails

```

We have seen that the largest value of n for which $W_n\mathbf{x} = \mathbf{b}$ can be solved accurately is 54, that means that for $n = 55$ the program returns an inaccurate value for the solution $\mathbf{x}$. In particular, instead of computing the value $\mathbf{x} = \mathbf{e} = [1, \dots, 1]^T$, it computes $\tilde{\mathbf{x}} = [1, \dots, 1, 0, 1]^T$. In other words we have:

$$\tilde{\mathbf{x}}_{54} = 0 \neq \mathbf{e}_{54} = 1. \tag{17}$$

In order to understand why this happens, we have verified that the matrices L_{55} and U_{55} are computed accurately. This is done in the function `compute_error_lufact_W(n)`, that can be found in the library Project_1. In this way, we know that the problem must be in the calculation of the forward and backward substitution. Note that having performed the LU factorization of the matrix W_{55} allows us to solve the system $W_{55}\mathbf{x} = \mathbf{b}_{55}$ by solving (in order) the following two linear systems with forward and backward substitutions:

$$\begin{cases} L\mathbf{y} = \mathbf{b} \\ U\mathbf{x} = \mathbf{y} \end{cases}, \tag{18}$$

where we named $L_{55} = L$, $U_{55} = U$, and $\mathbf{b}_{55} = \mathbf{b}$ for clarity. It can be verified that the analytical solution to the first system is $\mathbf{y} = [2^0 + 1, 2^1 + 1, \dots, 2^{n-2} + 1, 2^{n-1}]$. At this point, an important observation arises: if we consider $\mathbf{y}_{54} = 2^{53} + 1$, we may notice that, when the sum $2^{53} + 1$ is performed in double precision, the result is 2^{53}. This happens because, in double precision, $\text{fl}(2^{53} + 1) = 2^{53}$ or, in other words, 1 is smaller than the numerical distance between 2^{53} and the next representable number, when working in double precision. This can be justified by estimating the absolute error associated with the representation of $2^{53} + 1$:

$$|x - \text{fl}(x)| < u|x| = 2^{-53}(2^{53} + 1) = 1 + 2^{-53}. \tag{19}$$

After having computed $\mathbf{y}$, we can compute the solution of the second system starting from the bottom:

$$x_{55} = y_{55}/U_{55,55} = \frac{2^{54}}{2^{54}} = 1. \tag{20}$$

Now we update the vector $\mathbf{y}$ as follows:

$$\mathbf{y}^{(1)} = \mathbf{y} - x_{55}\mathbf{u}_{55}, \tag{21}$$

where $\mathbf{u}_{55}$ is the $55 - th$ and last column of U. The $54 - th$ component of $\mathbf{y}^{(1)}$ will be:

$$[\mathbf{y}^{(1)}]_{54} = [\mathbf{y}]_{54} - x_{55}U_{54,55} = 2^{53} + 1 - 2^{53}, \tag{22}$$

but, since in double precision we have $2^{53} + 1 = 2^{53}$, the returned value of $[\mathbf{y}^{(1)}]_{54}$ will be 0 and not 1. If we execute the same algorithm for bigger values of n, the number of elements of the solution vector that will be miscalculated will grow, starting from the penultimate element of the vector $\mathbf{x}$. It is worth noting that the last element of the solution is not affected by this kind of error because it is computed via the operation:

$$x_n = y_n/U_{n,n} = \frac{2^{n-1}}{2^{n-1}}, \tag{23}$$

that does not lead to catastrophic cancellations. These considerations imply that the GEPP algorithm is not backward stable when the input is a Wilkinson matrix. However, it is important to say that this is a very artificial example where the growth factor γ assumes the maximum possible value, i.e. $\gamma = 2^{n-1}$. However, in most cases, the GEPP algorithm is backward stable.

For the sake of completeness, we have developed a function, named `step_by_step_GEPP_W(n)` that calculates step by step the forward and backward substitution using the GEPP algorithm for Wikinson matrices, commented in such a way as to make each step explicit, with warnings that signal errors due to catastrophic cancellations.

3 Problem 3

Suppose that $A \in \mathbb{R}^{n\times n}$ is a nonsingular matrix and the LU factorization of A exists and has been computed. Consider two given vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$, we can define the matrix $\tilde{A} = A + \mathbf{u}\mathbf{v}^T$.

(1a) Prove that $\tilde{A}$ is nonsingular if and only if $\mathbf{v}^T A^{-1}\mathbf{u} \neq -1$.

Proof: We start proving that $\det(\tilde{A}) \neq 0$ implies that $\mathbf{v}^T A^{-1}\mathbf{u} \neq -1$. We can choose an orthonormal basis $\mathcal{B} = \{\mathbf{e}_i\}_{i=1,\dots,n}$ of $\mathbb{R}^n$ such that $\mathbf{u} = \alpha_1\mathbf{e}_1$ and $\mathbf{v} = \beta_1\mathbf{e}_1 + \beta_2\mathbf{e}_2$ with $\alpha_1, \beta_1, \beta_2 \in \mathbb{R}$. We can represent the matrix A with respect to the basis $\mathcal{B}$, denoting with $a_{ij} = \mathbf{e}_i^T A\mathbf{e}_j$ the element of the $i-th$ row and $j-th$ column of the matrix A written in the basis $\mathcal{B}$. We can represent the matrix $\mathbf{u}\mathbf{v}^T$ with respect to the basis $\mathcal{B}$, obtaining $\mathbf{u}\mathbf{v}^T = \alpha_1\beta_1\mathbf{e}_1\mathbf{e}_1^T + \alpha_1\beta_2\mathbf{e}_1\mathbf{e}_2^T$. It is known that we can compute the determinant of a $n \times n$ matrix M using the formula

$$\det(M) = \sum_{j=1}^{n} m_{ij}C_{ij}(M), \tag{24}$$

where C_{ij} is the cofactor of the element (i, j) of the matrix M. Therefore, the determinant of $\tilde{A}$ is:

$$\det(\tilde{A}) = \det(A) + \alpha_1\beta_1 C_{11}(A) + \alpha_1\beta_2 C_{12}(A). \tag{25}$$

Since we know that $\det(\tilde{A}) \neq 0$, we can write:

$$\det(A) + \alpha_1\beta_1 C_{11}(A) + \alpha_1\beta_2 C_{12}(A) \neq 0, \tag{26}$$

and, therefore:

$$\alpha_1\beta_1\frac{C_{11}(A)}{\det(A)} + \alpha_1\beta_2\frac{C_{12}(A)}{\det(A)} \neq -1, \tag{27}$$

where we divided for $\det(A)$ both sides of the equation, knowing that $\det(A) \neq 0$. At this point, it can be verified that

$$\mathbf{v}^T A^{-1}\mathbf{u} = \alpha_1\beta_1\frac{C_{11}(A)}{\det(A)} + \alpha_1\beta_2\frac{C_{12}(A)}{\det(A)}, \tag{28}$$

writing $\mathbf{u}$ and $\mathbf{v}$ in terms of $\mathbf{e}_1$ and $\mathbf{e}_2$ and $A^{-1} = \frac{1}{\det(A)}(cof(A))^T$, where $cof(A)$ is the matrix of cofactors of A. This proves that:

$$\mathbf{v}^T A^{-1}\mathbf{u} \neq -1. \tag{29}$$

In order to prove the converse implication, we consider again the expression for the determinant of $\tilde{A}$:

$$\det(\tilde{A}) = \det(A) + \alpha_1\beta_1 C_{11}(A) + \alpha_1\beta_2 C_{12}(A) = \det(A)\left(1 + \alpha_1\beta_1\frac{C_{11}(A)}{\det(A)} + \alpha_1\beta_2\frac{C_{12}(A)}{\det(A)}\right). \tag{30}$$

Here we can recognize the expression of $\mathbf{v}^T A^{-1}\mathbf{u}$, obtaining:

$$\det(\tilde{A}) = \det(A)(1 + \mathbf{v}^T A^{-1}\mathbf{u}). \tag{31}$$

Now, knowing that $\det(A) \neq 0$ and $\mathbf{v}^T A^{-1}\mathbf{u} \neq -1$, we obtain

$$\det(\tilde{A}) \neq 0 \tag{32}$$

and this concludes the proof. □

(1b) Show that:

$$\tilde{A}^{-1} = A^{-1} - \alpha A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}, \quad \text{where } \alpha = \frac{1}{\mathbf{v}^T A^{-1}\mathbf{u} + 1}. \tag{33}$$

Proof: We start noticing that the last expression is well defined since $\tilde{A}$ invertible implies $\mathbf{v}^T A^{-1}\mathbf{u} + 1 \neq 0$. Now we can manipulate the (33) multiplying both sides to the left by A and to the right by $\tilde{A}$, obtaining:

$$\begin{aligned} A &= \tilde{A} - \alpha\mathbf{u}\mathbf{v}^T A^{-1}\tilde{A} \\ &= A + \mathbf{u}\mathbf{v}^T - \alpha\mathbf{u}\mathbf{v}^T A^{-1}(A + \mathbf{u}\mathbf{v}^T) \\ &= A + (1-\alpha)\mathbf{u}\mathbf{v}^T - \alpha\mathbf{u}\mathbf{v}^T A^{-1}\mathbf{u}\mathbf{v}^T. \end{aligned} \tag{34}$$

Subtracting A from each side and dividing both sides by α (that is nonzero $\forall\ \mathbf{v}^T A^{-1}\mathbf{u} \in \mathbb{R} \setminus \{-1\}$) we obtain:

$$\mathbf{u}\mathbf{v}^T A^{-1}\mathbf{u}\mathbf{v}^T = \mathbf{u}\mathbf{v}^T(\alpha^{-1} - 1). \tag{35}$$

Finally, since $\mathbf{v}^T A^{-1}\mathbf{u} = \alpha^{-1} - 1$, the identity (35) is verified and this concludes the proof.

(1c) Assuming that LU factorization of A is already available, describe an $\mathcal{O}(n^2)$ algorithm to solve $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ for any right-hand side $\tilde{\mathbf{b}}$.
Supposing that $\tilde{A}$ is invertible, we can write the solution $\tilde{\mathbf{x}}$ using the Sherman-Morrison formula for $\tilde{A}$:

$$\tilde{\mathbf{x}} = \tilde{A}^{-1}\tilde{\mathbf{b}} = A^{-1}\tilde{\mathbf{b}} - \frac{A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}\tilde{\mathbf{b}}}{\mathbf{v}^T A^{-1}\mathbf{u} + 1}. \tag{36}$$

Algorithm:

- Compute $\mathbf{x}$ s.t. $A\mathbf{x} = \tilde{\mathbf{b}}$ and $\mathbf{y}$ s.t. $A\mathbf{y} = \mathbf{u}$ using backward and forward substitutions. This requires $\mathcal{O}(n^2)$ operations.
- Compute $\gamma = \frac{\mathbf{v}^T\mathbf{x}}{\mathbf{v}^T\mathbf{y}+1}$. This requires $\mathcal{O}(n)$ operations.
- Compute $\tilde{\mathbf{x}} = \mathbf{x} - \gamma\mathbf{y}$. This requires $\mathcal{O}(n)$ operations.

(2) Assuming again that the LU factorization of A existes and has been computed, describe an efficient algorithm for solving the *bordered system*

$$\begin{bmatrix} A & \mathbf{u} \\ \mathbf{v}^T & \beta \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ z \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ c \end{bmatrix}, \tag{37}$$

where z is unknown and β and c are given scalars. When does this system have a unique solution?
Solution: Defining

$$A' = \begin{bmatrix} A & \mathbf{u} \\ \mathbf{v}^T & \beta \end{bmatrix}, \mathbf{x}' = \begin{bmatrix} \mathbf{x} \\ z \end{bmatrix}, \mathbf{b}' = \begin{bmatrix} \mathbf{b} \\ c \end{bmatrix}, \tag{38}$$

the system above rewrites as:

$$A'\mathbf{x}' = \mathbf{b}'. \tag{39}$$

The LU factorization of the matrix $A' = L'U'$ exists and the matrices L' and U' take the following form:

$$L' = \begin{bmatrix} L & \mathbf{0} \\ \mathbf{f}^T & 1 \end{bmatrix}, U' = \begin{bmatrix} U & \mathbf{g} \\ \mathbf{0} & \gamma \end{bmatrix}. \tag{40}$$

In order to get the values of $\mathbf{f}, \mathbf{g} \in \mathbb{R}^n$ and $\gamma \in \mathbb{R}$, we impose $A' = L'U'$, obtaining the following system:

$$\begin{cases} L\mathbf{g} = \mathbf{u} \\ U^T\mathbf{f} = \mathbf{v} \\ \mathbf{f}^T\mathbf{g} + \gamma = \beta \end{cases}. \tag{41}$$

Here we can find $\mathbf{f}$ and $\mathbf{g}$ with forward substitutions with $\mathcal{O}(n^2)$ operations. Therefore, we can rewrite the last equation as:

$$\gamma = \beta - \mathbf{v}^T A^{-1}\mathbf{u}. \tag{42}$$

In order to impose that the bordered system has a unique solution, we have to require that $\det(A') \neq 0$, that is true if and only if all the diagonal elements of U' are nonzero and, given that $\det(A) \neq 0$, this means requiring that $\gamma \neq 0$. Therefore, the condition for the uniquenes of the solution becomes:

$$\gamma = \beta - \mathbf{v}^T A^{-1}\mathbf{u} \neq 0 \Rightarrow \quad \mathbf{v}^T A^{-1}\mathbf{u} \neq \beta. \tag{43}$$