Mathematical models for economic applications

July 16, 2024

1 Mathematical models for economic applications

Owner: Thomas De Massari
e-mail: thomas.demassari@gmail.com
Linkedin: https://www.linkedin.com/in/thomasdemassari/
GitHub: https://github.com/thomasdemassari/

During the last semester of my BSc (February-May 2024), I attended the Mathematical Models for Economic Applications course, held by Professor Silvia Bortot and Professor Ricardo Alberto Marques Pereira. In this file, you will find functions that I created to solve exercises related to the topics covered in class. The primary goal of these functions is to improve my Python programming skills while studying for the final exam. Furthermore, I am fully aware that there might be more efficient ways to implement some of the algorithms and more efficient Python functions, but the purpose of these scripts is to solve the course exercises in Python. For any questions or requests for information, please contact me at thomas.demassari@gmail.com

Main topics of the course:
- Introductory Linear Algebra
- Linear Programming (Simplex Algorithm and Knapsack Problem)
- Dynamic Models (Consensus Dynamics and Linear Dynamic Population Redistribution).

1.1 Libraries

[ ]:
```
import numpy as np
import math
import pandas as pd
import scipy

from itertools import combinations

import warnings
warnings.simplefilter(action = "ignore", category = FutureWarning)

from datetime import datetime
print(f"Last update: {datetime.now().replace(second = 0, microsecond = 0)}")
```

```
Last update: 2024-07-16 07:47:00
```

1.2 Introductory matrix algebra

- Matrix addition
- Matrix multiplication
- Determinant of a matrix
- Inverse matrix
- Minor of a matrix
- Rank of a matrix
- Gaussian elimination method for solving a linear system
- Right and left eigenvectors

1.2.1 Matrix addition

[ ]:
```
def matrixplusmatrix(matrix1, matrix2):
    try:
        row1 = matrix1.shape[0]                                  # Number of␣
↪rows of the first matrix
        col1 = matrix1.shape[1]                                  # Number of␣
↪columns of the first matrix
        row2 = matrix2.shape[0]                                  # Number of␣
↪rows of the second matrix
        col2 = matrix2.shape[1]                                  # Number of␣
↪columns of the second matrix
        if (row1 == row2) and (col1 == col2):                    # The two␣
↪matrixs must have the same dimensions
            result = np.zeros((row1, col1))
            for i in range(row1):
                for j in range(col1):
                    result[i][j] = matrix1[i][j] + matrix2[i][j]
            return result
        else:
            raise Exception("The two matrixs have different dimensions.")
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.2.2 Matrix multiplication

[ ]:
```
def matrixtimesmatrix(matrix1,matrix2):
    try:
        row1 = matrix1.shape[0]                         # Number of rows of the␣
 ↪first matrix
        col1 = matrix1.shape[1]                         # Number of columns of␣
 ↪the first matrix
        row2 = matrix2.shape[0]                         # Number of rows of the␣
 ↪second matrix
        col2 = matrix2.shape[1]                         # Number of columns of␣
 ↪the second matrix

        if (col1 == row2):                              # The two matrices must␣
 ↪be conformable
            result = np.zeros((row1, col2))
            for i in range(row1):
                for j in range(col2):
                    for k in range(col1):
                        tmp = matrix1[i][k]*matrix2[k][j]
                        result[i][j] = result[i][j] + tmp
            return result
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
 ↪for the parameters")
```

1.2.3 Determinant of a matrix

[ ]:
```
def determinant(matrix):
    try:
        row = matrix.shape[0]                                                  ␣
 ↪            # Number of rows of the first matrix
        col = matrix.shape[1]                                                  ␣
 ↪            # Number of columns of the first matrix

        result = 0
        if (row == col):
            if row == 1:
                result = matrix[0][0]
                result = round(result, 2)
                return result
            else:
                if row == 2:
                    result = ((matrix[0][0]*matrix[1][1]) -␣
 ↪(matrix[0][1]*matrix[1][0]))
                    result = round(result, 2)
                    return result
```

```python
                else:
                    if row == 3:                                                                # Rule of Sarrus
                        col0 = matrix[:,0]
                        col1 = matrix[:,1]
                        sarrus_matrix = np.concatenate((matrix, col0[:, np.newaxis], col1[:, np.newaxis]), axis=1)
                        i = 0
                        j = 0
                        for i in range(3):
                            tmp1 = sarrus_matrix[0][i]*sarrus_matrix[1][i+1]*sarrus_matrix[2][i+2]
                            result = result + tmp1
                        for j in range(3):
                            tmp2 = (sarrus_matrix[2][j]*sarrus_matrix[1][j+1]*sarrus_matrix[0][j+2])
                            result = result - tmp2
                        result = round(result, 2)
                        return result
                    else:                                                                       # Laplace expansion (row = 0)
                        j = 0
                        for j in range(col):
                            minor = np.delete(np.delete(matrix, 0, axis=0), j, axis=1)
                            complementary_minor =  np.linalg.det(minor)
                            tmp = matrix[0][j]*((-1)**j)*complementary_minor
                            result = result + tmp
                        result = round(result, 2)
                        return result
        else:
            print("The provided input is not a square matrix.")
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values for the parameters")
```

1.2.4 Inverse matrix

[ ]:
```python
def inversematrix(matrix):
    try:
        row = matrix.shape[0]                                                               # Number of rows of the first matrix
        col = matrix.shape[1]                                                               # Number of columns of the first matrix

        if (row == col):
            det = determinant(matrix)
```

```
            inverse_matrix = np.zeros((row, col))
            if det != 0:
                for i in range(row):
                    for j in range(col):
                        complementary_minor = np.linalg.det(np.delete(np.
↪delete(matrix, j, axis=0), i, axis=1))
                        complementary_algebraic =␣
↪((-1)**(j+i))*complementary_minor
                        tmp = complementary_algebraic/det
                        inverse_matrix[i][j] = tmp
                return inverse_matrix
            else:
                print("The determinant of the matrix is zero. The inverse␣
↪matrix does not exist.")

        else:
            print("The provided input is not a square matrix.")

    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.2.5 Minor of a matrix

```
[ ]: def minor(matrix, show = False, order = "all"):
    try:
        row = matrix.shape[0]
        col = matrix.shape[1]
        nminors_max = min(row, col)
        rows = list(range(row))
        cols = list(range(col))

        n_minors = 0
        k = 1
        for k in range(1, nminors_max+1):
            n_minors = math.comb(row, k)*math.comb(col, k) + n_minors

        # All possible combinations between rows and columns
        all_combinations = list()
        # Add all possible combinations between vector1 and vector2
        for length in range(1, min(len(rows), len(cols)) + 1):
            for combination1 in combinations(rows, length):
                for combination2 in combinations(cols, length):
                    combination1 = list(combination1)
                    combination2 = list(combination2)
                    all_combinations.append((combination1, combination2))
        # Add combinations with different lengths between vector1 and vector2
```

```python
        for combination1 in combinations(rows, len(rows)):
            for combination2 in combinations(cols, len(cols)):
                combination1 = list(combination1)
                combination2 = list(combination2)
                all_combinations.append((combination1, combination2))

        # Consider only the combinations with the same length (squared matrixs)
        vector_of_combinations = list()
        for i in range(len(all_combinations)):
            if len(all_combinations[i][0]) == len(all_combinations[i][1]):
                vector_of_combinations.append(all_combinations[i])

        # Rows and cols of the minors
        m = [vector_of_combinations[i][0] for i in␣
↪range(len(vector_of_combinations))]
        n = [vector_of_combinations[j][1] for j in␣
↪range(len(vector_of_combinations))]

        counter = 0
        list_of_minors = list()

        for i in range(len(m)):
                output = (matrix[m[i]][:, n[i]])
                list_of_minors.append(output)
                counter = counter + 1

        if (order == "all"):
            if show == True:
                print(f"Minor of order: {len(m[i])}:\n{output}")
                if counter == n_minors:
                    print(f"All minors ({counter}) have been calculated.")
                else:
                    print(f"Not all minors have been calculated. Only {counter}␣
↪of {n_minors} have been calculated.")
            # list_of_minors = np.array(list_of_minors)
            return list_of_minors
        else:
            list_of_k_order_minors = list()
            for z in range(len(list_of_minors)):
                if len(list_of_minors[z]) == order:
                    list_of_k_order_minors.append((list_of_minors[z]))
            # list_of_k_order_minors = np.array(list_of_k_order_minors)
            return list_of_k_order_minors

    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.2.6 Rank of a matrix

[ ]:
```
def rank(matrix):
    try:
        row = matrix.shape[0]
        col = matrix.shape[1]

        max_order = min(row, col)

        index = list(range(1, max_order+1))
        index.reverse()

        rank_of_the_matrix = 0
        k = 1
        for k in index:
            minors = minor(matrix, order = k)
            minors = np.array(minors)
            list_tmp = list()
            for i in range(minors.shape[0]):
                tmp = determinant(minors[i])
                list_tmp.append(tmp)
                if all(element == 0 for element in list_tmp):
                    rank_of_the_matrix = k - 1
        return rank_of_the_matrix

    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.2.7 Gaussian elimination method for solving a linear system

[ ]:
```
def check_diagonal(matrix):
    try:
        for i in range(len(matrix)):
            for j in range(len(matrix[0])):
                if i > j and matrix[i][j] != 0:
                    return False
        return True
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

[ ]:
```
def guassian_elimination(matrixA, matrixB):
    try:
        np.seterr(divide='ignore', invalid='ignore')                          ␣
↪                        #ignore the warning of division by zero
```

```
    matrixAB = np.concatenate((matrixA, matrixB), axis=1)

    rowA = int(matrixA.shape[0])
    colA = int(matrixA.shape[1])

    target = np.zeros((rowA, colA+1))
    final_target = np.zeros((rowA, colA+1))

    if rank(matrixA) == rank(matrixAB):
        #Gaussian elimination
        index1 = list(range(rowA))
        index2 = list(range(colA))

        for j in range(colA):
            if matrixAB[0][0] != 1 or matrixAB[0][0] != 0:
                target[0][j] = matrixAB[0][j]/matrixAB[0][0]
            else:
                target[0][j] = matrixAB[0][j]


        # Find the indices of the first non-zero element in each row
        row_indices, col_indices = np.nonzero(matrixA)
        # Filter only the indices of the first occurrence of each row
        unique_row_indices, first_col_indices = np.unique(row_indices, return_index=True)
        # Get the indices of the first non-zero element in each row
        first_nonzero_indices = (unique_row_indices, col_indices[first_col_indices])                #rows in the first element, cols in the second element

        i = 1
        j = 0
        for i in range(1, rowA):
            index_col = first_nonzero_indices[1][i]
            coeff = (matrixAB[i][index_col]/matrixAB[i-1][index_col])

            for j in range(colA + 1):
                    if matrixAB[i][j] != 0:
                        target[i][j] = matrixAB[i][j] - (matrixAB[i-1][j]*coeff)
                        tmp = False
                    else:
                        target[i][j] = matrixAB[i][j]
                        tmp = False

            correctness = check_diagonal(target)
            if correctness == True:
```

```
                    break
                else:
                    continue

            i = 1
            j = 0

            for j in range(colA +1):
                final_target[0][j] = target[0][j]

            for i in range(1, rowA):
                for j in range(colA + 1):
                    if target[i][i] != 1 or target[i][i] != 1:
                        final_target[i][j] = target[i][j]/target[i][i]
                    else:
                        final_target[i][j] = target[i][j]

            final_target[np.isnan(final_target)] = 0
            return final_target

        else:
            print("The system has no solution.")
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.2.8 Right and left eigenvectors

[ ]:
```
def eigeinvectors_calculator(matrix, side = "right"):
    try:
        eingenvalues = np.linalg.eigvals(matrix)
        I_matrix = np.eye(len(matrix[0]), len(matrix[:, 0]))                 ␣
↪                              # Identity matrix

        result = pd.DataFrame(columns = ["Eigenvalue", "Eigenvector"])

        for i in range(len(eingenvalues)):
            eingenvalue = round(eingenvalues[i], 5)
            a = matrix - (eingenvalue * I_matrix)                                ␣
↪                              # (A - lamba*I)
            b = np.zeros((len(matrix[:, 0]), 1))                                 ␣
↪                              # 0, 0, ..., 0

            # Right eigenvector
            if side == "right":
                # Solve the linear system x*(A-lambda*I) = 0
```

```
                x = scipy.linalg.null_space(a)                                    ␣
↪                               # x1, x2, ..., x_n, i.e the eigenvector

                # Check the solution of the linear system
                if np.allclose(np.dot(a, x), b) == False:
                    raise Exception("ERROR: The function get wrong")

            else:
                # Left eigenvector
                if side == "left":
                    # Solve the linear system (A-lambda*I)T * x = 0 (which is␣
↪equivalent to solving xT * (A-lambda*I) = 0)
                    x = scipy.linalg.null_space(a.T)

                    # Check the solution of the linear system
                    if np.allclose(np.dot(a.T, x), b) == False:
                        raise Exception("ERROR: The function get wrong")

                else:
                    raise Exception("ERROR: Please pass a valid value for the␣
↪parameter 'side' (either 'left' or 'right').")

            # Save the result
            x = np.round(x, 5)
            newrow = {"Eigenvalue": [eingenvalue], "Eigenvector": [x]}
            result.loc[len(result)] = newrow

        return result
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

[ ]:

```
def solve_linearsystem(matrixA, matrixB, n_unknowns = 0,␣
↪show_number_and_type_of_solutions = False):
#NB this function doesn't provide the numerical solution of the linear system,␣
↪but provides only the linear equations to solve the system.
    try:
        reduced_form_matrix = guassian_elimination(matrixA, matrixB)

         #Matrix with only the basics varaibles
        non_zero_mask = ~np.all(reduced_form_matrix == 0, axis=1)
        matrix_no0 = reduced_form_matrix[non_zero_mask]

        row = reduced_form_matrix.shape[0]
        col = reduced_form_matrix.shape[1]

        row_no0 = matrix_no0.shape[0]
```

```
        col_no0 = matrix_no0.shape[1]

        #number of solutions and type of variables
        if show_number_and_type_of_solutions == True:
            basicsvar = list()
            freevar = list()
            if n_unknowns == 0:
                print("Please provide the number of unknowns.")
            else:
                if (n_unknowns - rank(matrixA) == 0):
                    print("The system has a unique solution:\n")
                else:
                    print(f"The system has ∞^{n_unknowns - rank(matrixA)}␣
↪solutions:\n")
            #Basics and free variables
            for i in range(row):
                tmp = f"x{i+1}"
                if reduced_form_matrix[i][i] == 1 or reduced_form_matrix[i][i]␣
↪== -1:
                    basicsvar.append(tmp)
                else:
                    freevar.append(tmp)
            print(f"Basics variables: {basicsvar}\nFree variables: {freevar}\n")

        #Modify the matrix with the names of the variables
        matrix_with_varnames = np.empty((row, col), dtype=object)
        for i in range(row_no0):
            for j in range(col_no0-1):
                matrix_with_varnames[i][j] =␣
↪f"{reduced_form_matrix[i][j]}x{j+1}"
        for i in range(row):
            matrix_with_varnames[i][-1] = reduced_form_matrix[i][-1]

        # print("Final result:")
        final_result = list()
        for i in range(row_no0):
            tmp = f"x{i+1} = {matrix_with_varnames[i][i+1:]}"
            final_result.append(tmp)
            # print(tmp)
        return final_result

    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.3 Linear programming

- Simplex Algotithm

- I/O Knapsack Problem using Branch and Bound Algotithm

1.3.1 Simplex Algorithm

The following function solves problems of minimum and maximum using the Simplex Algorithm. It takes the following inputs:
- *function_to_minimax* (as a np.array), containing the function to maximize (minimize);
- *constraints* (as a np.array), containing the constraints of the problem;
- *goal* (as a string). If it is a maximum problem, goal is set to "max"; if it is a minimum problem, goal is set to "min".

The *SimplexAlgorithm* function calls four other functions:
- *entryANDexit_criteria*, to find a new base, according to Simplex Algorithm. It returns two lists, one with indices and one with names of base variables;
- *MoveToAnotherVertex*, to compute the right matrix according to the new base variables, calculated with this function;
- *Matrix_Primal2DualProbelm*, to calculate the matrix of the dual problem;
- *Solution_Dual2PrimalProblem*, to find the solution of the primal problem, given the solution of the dual one.

Example
Given the problem:

$$max \;\; 6x_1 + 2x_2 + 4x_3 = 0$$

$$s.t. \begin{cases} x_1 + 2x_2 + 3x_3 \leq 60 \\ 2x_1 + x_2 + x_3 \leq 30 \end{cases}$$

The *SimplexAlgorithm* function takes as inputs:

```
function_to_minimax = np.array([[6, 2, 4, 0]])
constraints = np.array([[1, 2, 3, 60],
                        [2, 1, 1, 30]])
```

And the output will be:

```
                                         0
max value of the function            108.0
Names of base variables           [x1, x3]
Values of base variables       [6.0, 18.0]
```

[ ]:
```
def entryANDexit_criteria(complete_matrix, constraints_index,␣
↪names_of_variables, names_of_constraints, goal = "max"):
    """
    This function takes the complete_matrix, constraints_index,␣
↪names_of_variables, names_of_constraints and goal (all computed by the␣
↪SimplexAlgorithm function) as parameters and returns a new set of base␣
↪variables (and relative names) according to Simplex Algorithm.
```

```
    NB: I am fully aware know that the nomenclature "constraints_index" to␣
↪indicate the indices of base variables could be tricky, but it was useful to␣
↪me when I wrote the first version of this code.
    """

    try:
        # Working matrix to pd.Dataframe
        # Names of rows and cols
        rows = names_of_constraints + ["z"]
        cols = names_of_variables + ["value"]
        df_working_matrix = pd.DataFrame(complete_matrix, index = rows, columns␣
↪= cols)

        z_lines_complete_matrix = list(complete_matrix[-1, :][:-1])                 ␣
↪                           # Line of function to minimax



        # ENTRY CRITERIA
        entry_criteria_index_list = list()                                          ␣
↪                           # Where I will save the index of possible values␣
↪of the entry criteria
        entry_criteria_values_list = list()                                         ␣
↪                           # Where I will save the possible values of entry␣
↪criteria
        entry_criteria_names_list = list()                                          ␣
↪                           # Where I will save the possible names of␣
↪variables of entry criteria

        for i in range(len(z_lines_complete_matrix)):
            if i in constraints_index:                                              ␣
↪                           # constraints_index contains the indices of the␣
↪positions of the base variables
                continue
            else:
                if goal == "max":
                    if z_lines_complete_matrix[i] < 0:
                        entry_criteria_index_list.append(i)                         ␣
↪                           # Save the index
                        entry_criteria_values_list.
↪append(abs(z_lines_complete_matrix[i]))               # Save the abs of the␣
↪value
                        entry_criteria_names_list.append(names_of_variables[i])␣
↪                           # Save the name of the variable
                if goal == "min":
                    raise Exception("Something went wrong. This function solves␣
↪minimization problems using the Duality Theorem.")
```

```
        entry_criteria_selection_value = max(entry_criteria_values_list)                                                         # Value of the variable that will entry
        entry_criteria_index = entry_criteria_index_list[entry_criteria_values_list.index(entry_criteria_selection_value)] # Index of the variable that will entry
        entry_criteria_name = entry_criteria_names_list[entry_criteria_values_list.index(entry_criteria_selection_value)]  # Name of the variable that will entry



        # EXIT CRITERIA
        col_of_entry_criteria = complete_matrix[:, entry_criteria_index][:-1]                                 # Column where select the exit value
        col_of_constraints_bound = complete_matrix[:, -1][:-1]                                                  # Column of values of constraints (i.e. col where there are 60 in constraint x1 - x2 < 60)

        exit_criteria_values_list = list()                                                       # Where I will save values of possibile of exit value
        exit_criteria_indexs_list = list()                                                       # Where I will save the index of possible values of the entry criteria

        for j in range(len(col_of_entry_criteria)):
            if col_of_entry_criteria[j] > 0:
                exit_ratio_tmp = col_of_constraints_bound[j]/col_of_entry_criteria[j]                          # b_i/a_ij
                exit_criteria_values_list.append(exit_ratio_tmp)                                 # Save the value, if non-negative
                exit_criteria_indexs_list.append(j)                                              # Save the respective index

        if len(exit_criteria_indexs_list) == 0:
            constraints_index = "Unbounded Optimal Solution"
            names_of_constraints = "Unbounded Optimal Solution"
            return constraints_index, names_of_constraints

        exit_criteria_value = min(exit_criteria_values_list)                                     # Value of the variable that will exit
```

```
        exit_criteria_name = df_working_matrix.
↪index[exit_criteria_indexs_list[exit_criteria_values_list.
↪index(exit_criteria_value)]] # Name of the variable that will exit

        exit_criteria_index = df_working_matrix.columns.
↪get_loc(exit_criteria_name)                        # Index of the variable␣
↪that will exit

        # FIND THE NEW BASE VARIABLES
        exit_index_in_constraints_index_list = constraints_index.
↪index(exit_criteria_index)                # Index in constraints_index of␣
↪the exit variable
        constraints_index[exit_index_in_constraints_index_list] =␣
↪entry_criteria_index
        constraints_index.sort()

        names_of_constraints[exit_index_in_constraints_index_list] =␣
↪entry_criteria_name
        names_of_constraints = sorted(names_of_constraints, key=lambda x:␣
↪names_of_variables.index(x))

        return constraints_index, names_of_constraints
    except:
        raise Exception("ERROR: Something went wrong in the␣
↪entryANDexit_criteria. Please check the passed values.")
```

[ ]:

```
def MoveToAnotherVertex(complete_matrix, constraints_index):
    """
    This function takes the complete_matrix and the constraints_index (both␣
↪computed by the SimplexAlgorithm function) as parameters and returns a new␣
↪matrix based on the index of base variables (constraints_index) that are␣
↪passed, according to Simplex Algorithm.
    NB: I am fully aware that the nomenclature "constraints_index" to indicate␣
↪the indices of base variables could be tricky, but it was useful to me when␣
↪I wrote the first version of this code.
    """
    try:
        tmp_matrix = np.zeros(np.shape(complete_matrix))                        ␣
↪                        # Empty matrix where I will save the new␣
↪complete_matrix
        tmp_constraints_value_for_matrix = np.eye(len(complete_matrix[:,0]),␣
↪len(constraints_index))     # Diagonal matrix
        target_matrix = np.zeros(np.shape(complete_matrix))                     ␣
↪                        # Target matrix. If I will do all right, tmp_matrix␣
↪will equal to target_matrix
```

```
        for i in range(len(constraints_index)):
            den_to_one = complete_matrix[i, constraints_index[i]]                  ␣
↪                       # Value to have 1 in position (i,␣
↪constraints_index[i])
            if den_to_one != 0:
                for j in range(len(complete_matrix[i,:])):
                    tmp_matrix[i, j] = complete_matrix[i, j] / den_to_one
            else:
                raise Exception("Unhandled error.")                                ␣
↪                       # If the code will go here means that something went␣
↪wrong in the Simplex Algorithm

            for index_addtozero_rows in range(len(complete_matrix[:, 0])):
                if i != index_addtozero_rows:                                      ␣
↪                       # If the row i is different than␣
↪index_addtozero_rows means that we are in a line where we want a 0 under the␣
↪base variable constraints_index[i]
                    add_to_zero = (complete_matrix[index_addtozero_rows,␣
↪constraints_index[i]]) / (complete_matrix[i, constraints_index[i]])

                    for index_addtozero_cols in range(len(complete_matrix[0, :
↪])):
                        tmp = (complete_matrix[i, index_addtozero_cols] * (-␣
↪add_to_zero)) + complete_matrix[index_addtozero_rows, index_addtozero_cols]
                        tmp_matrix[index_addtozero_rows, index_addtozero_cols]␣
↪= tmp

            complete_matrix = tmp_matrix

        # Complete the target matrix and control if tmp_matrix equals to␣
↪target_matrix
        for j in range(len(complete_matrix[0,:])):
            if j in constraints_index:
                target_matrix[:, j] = tmp_constraints_value_for_matrix[:,␣
↪constraints_index.index(j)]
            else:
                target_matrix[:, j] = complete_matrix[:, j]

        control = list()
        for j in range(len(complete_matrix[0,:])):
            if j in constraints_index:
                if np.array_equal(complete_matrix[:,j], tmp_matrix[:,j]):
                    control.append(1)                                              ␣
↪                        # 1 means True
                else:
```

```
                    control.append(0)                                                    ␣
↪                           # 0 means True

        if sum(control) == 0:
            raise Exception("ERROR: Something went wrong in the function␣
↪MoveToAnotherVertex.")
        else:
            return complete_matrix
    except:
        raise Exception("ERROR: Something went wrong in the MoveToAnotherVertex.
↪ Please check the passed values.")
```

[ ]:
```
def Matrix_Primal2DualProblem(function_to_minimax, constraints):
    """
    This function takes the parameters of the function to maximize (minimize)␣
↪and the coefficients of constraints, both of type np.array. The last column␣
↪of both arrays contains the bounds/values of the constraint/function.␣
↪Primal2DualProble, returns the Dual Problem matrix (type: np.array) and the␣
↪indices of base variables.
    NB: I am fully aware that the nomenclature "constraints_index" for␣
↪indicating the indices of base variables could be tricky, but it was useful␣
↪to me when I wrote the first version of this code.

    Example:
    INPUT:
        max         z = 2x_1 + 3x_2 + 12x_3
        subject to 4x_1 + 9x_2 + 0x_3 < 4
                   5x_1 + 0x_2 + 1x_3 < 9

        function_to_minimax = np.array([[2, 3, 12, 0]])
        constraints         = np.array([[4, 9, 0 ,4],
                                        [5, 0, 1, 9]])

    OUTPUT
    new_working_matrix      = np.array([[4, 9, 0, 1, 0 ,4],
                                        [5, 0, 1, 0, 1, 9],
                                        [2, 3, 12, 0, 0, 0]])
    new_constraints_index   = [3, 4]
    """
    try:
        tmp_working_matrix = np.vstack((constraints, function_to_minimax))
        new_working_matrix = np.zeros(np.shape(tmp_working_matrix.T))

        # From Primal Matrix Probelm to Dual Matrix Problem
        for i in range(len(new_working_matrix[:, 0])):
            new_working_matrix[i, :] = tmp_working_matrix[:, i]
```

```
        new_working_matrix[-1,:] = -new_working_matrix[-1,:]
        new_constraints_index = list(range(len(new_working_matrix[0,:]) - 1,␣
↪(len(new_working_matrix[0,:]) - 1 + len(new_working_matrix[:,0]) - 1)))

        return new_working_matrix, new_constraints_index
    except:
        raise Exception("ERROR: Something went wrong in the␣
↪Matrix_Primal2DualProbelm. Please check the passed values.")
```

[ ]:
```
def Solution_Dual2PrimalProblem(constraints, names_of_variables,␣
↪names_of_variables_in_the_dual_solution,␣
↪values_of_variables_in_the_dual_solution):
    """
    This function takes the solutions of the Dual Problem and returns the␣
↪solutions of the Primal one. It is necessary because my SimplexAlgorithm␣
↪function uses the Duality Theorem to solve minimum problems.
    NB: I am fully aware that the nomenclature "constraints_index" to indicate␣
↪the indices of base variables could be tricky, but it was useful to me when␣
↪I wrote the first version of this code.
    """
    try:
        # Variables
        # MAX Problem (Dual)
        dual_names_of_variables = names_of_variables
        dual_names_of_constraints = names_of_variables_in_the_dual_solution
        dual_values_of_variables = values_of_variables_in_the_dual_solution
        # dual_intial_constraints_index = initial_constraints_index
        # MIN Probelm (Primal)
        primal_constraints = constraints


        # From the Dual to the Primal solutions
        primal_names_of_variables = list()
        for i in range(len(dual_names_of_variables)):
            var_dual = dual_names_of_variables[i]
            xs = var_dual[0]
            n = var_dual[1:]

            if xs == "x":
                xs = "s"
            else:
                xs = "x"

            var_dual = xs + n
            primal_names_of_variables.append(var_dual)

        # Sorted primal_names_of_variables and dual_names_of_variables
```

```python
        # primal_names_of_variables
        def custom_sort_x(s):
            if "x" in s:
                return 0
            elif "s" in s:
                return 1
            else:
                return 2
        primal_names_of_variables_sorted = sorted(primal_names_of_variables,␣
↪key = custom_sort_x)
        # dual_names_of_variables
        def custom_sort_s(s):
            if "s" in s:
                return 0
            elif "x" in s:
                return 1
            else:
                return 2
        dual_names_of_variables_inverted = sorted(dual_names_of_variables, key␣
↪= custom_sort_s)


        # Creating a dictionary, where keys are duals names of variables
        primal2dual_vars_dict = {}
        for x, y in zip(primal_names_of_variables_sorted,␣
↪dual_names_of_variables_inverted):
            primal2dual_vars_dict[x] = y

        # Creating the working matrix (of the Primal Problem)
        zeros_for_primal_matrix = np.eye(len(primal_constraints[:, 0]))
        constraints_bounds_of_primal_problem = primal_constraints[:, [-1]]
        primal_working_matrix = primal_constraints[:, :-1]
        primal_working_matrix = np.hstack((primal_working_matrix,␣
↪-zeros_for_primal_matrix))
        primal_working_matrix = np.hstack((primal_working_matrix,␣
↪constraints_bounds_of_primal_problem))

        # Complete the base variable
        dual_base_vars = list()
        counter_values_of_constraints = 0
        for i in range(len(dual_names_of_variables)):
            if dual_names_of_variables[i] in dual_names_of_constraints:
                dual_base_vars.
↪append(dual_values_of_variables[counter_values_of_constraints])
                counter_values_of_constraints += 1
            else:
```

```
                dual_base_vars.append(0)

        # Applying Strong Duality Thoerem
        tmp_index_of_rows = 0
        for i in range(len(dual_names_of_variables)):
            if dual_base_vars[i] != 0:
                tmp_index_of_col = dual_names_of_variables_inverted.
↪index(primal2dual_vars_dict[primal_names_of_variables[i]])
                primal_working_matrix[:, tmp_index_of_col] = 0
                tmp_index_of_rows += 1

        # Linear sistem (from Dual to Primal)
        A = primal_working_matrix[:, :-1]                                          ␣
↪                                                       # Coefficients
        b = primal_working_matrix[:, -1]                                           ␣
↪                                                       # Value

        x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)                  ␣
↪                                                       # Solution of the linear␣
↪system Ax = b

        # Find the solution of the Primal Problem
        values_of_variables_in_the_dual_solution = list()
        names_of_variables_in_the_dual_solution = list()
        for i in range(len(x)):
            if x[i] != 0:
                values_of_variables_in_the_dual_solution.append(round(x[i], 5))
                names_of_variables_in_the_dual_solution.
↪append(primal_names_of_variables_sorted[i])

        return values_of_variables_in_the_dual_solution,␣
↪names_of_variables_in_the_dual_solution
    except:
        raise Exception("ERROR: Something went wrong in the␣
↪Solution_Dual2PrimalProblem. Please check the passed values.")
```

[ ]:
```
def SimplexAlgorithm(function_to_minimax, constraints, goal = "max"):
    """
    This function takes the parameters of the function to maximize (minimize),␣
↪the coefficients of constraints, both of type np.array, and whether it is a␣
↪problem of maximization (goal = 'max') or minimization (goal = 'min'). The␣
↪last column of both arrays contains the bounds/values of the constraint/
↪function. The SimplexAlgorithm returns a Pandas DataFrame object which␣
↪contains the maximum (minimum) value of the function, and the names and␣
↪values of the base variables. Minimum problems are solved using the Duality␣
↪Theorem.
```

```
    NB: I am fully aware that the nomenclature "constraints_index" to indicate␣
↪the indices of base variables could be tricky, but it was useful to me when␣
↪I wrote the first version of this code.

    Example:
    INPUT:
        max        z = 2x_1 + 3x_2 + 12x_3
        subject to 4x_1 + 9x_2 + 0x_3 < 4
                   5x_1 + 0x_2 + 1x_3 < 9

        function_to_minimax = np.array([[2, 3, 12, 0]])
        constraints         = np.array([[4, 9, 0 ,4],
                                        [5, 0, 1, 9]])
        goal                = 'max'

    OUTPUT
    result = max value of the function          float
             Names of base variables      [var1, var2]
             Values of base variables     [int1, int2]
    """
    try:
        initial_goal = goal                                                     ␣
↪                        # Save here the initial goal of the problem

        # Creating the working matrix
        if goal == "max":
            working_matrix = np.vstack((constraints, -function_to_minimax))
            constraints_index = list(range(len(working_matrix[0,:]) - 1,␣
↪(len(working_matrix[0,:]) - 1 + len(working_matrix[:,0]) - 1)))
        else:
            if goal == "min":
                working_matrix, constraints_index =␣
↪Matrix_Primal2DualProblem(function_to_minimax, constraints)
                goal = "max"
            else:
                raise Exception("ERROR: Please pass a valid value for the␣
↪parameter goal ('max' or 'min')")

        # Constraints saturation
        zeros_of_constraints = np.eye(len(working_matrix[:, 0]) -1)
        zeros_of_function_to_minimax = np.zeros((1, len(zeros_of_constraints[0,:
↪])))
        zeros = np.vstack((zeros_of_constraints, zeros_of_function_to_minimax))

        constraints_bounds = working_matrix[:, [-1]]                            ␣
↪                        # Constraint bounds and function value
```

```python
        working_matrix = working_matrix[:, :-1]                         # Coefficients of constraints and function

        working_matrix = np.hstack((working_matrix, zeros))
        working_matrix = np.hstack((working_matrix, constraints_bounds))                        # Definitive working matrix

        # Names of variables and constraints
        names_of_variables = list()                         # Where I will save the names of variables
        names_of_constraints = list()                       # Where I will save the names of base variables
        counter_s = 1                                       # Counter to count initial slacks
        counter_x = 1                                       # Counter to count intial variables
        for i in range(len(working_matrix[0,:]) - 1):
            if i in constraints_index:
                names_of_variables.append(f"s{counter_s}")
                names_of_constraints.append(f"s{counter_s}")
                counter_s += 1
            else:
                names_of_variables.append(f"x{counter_x}")
                counter_x += 1

        # Shortly function to order names_of_variables and names_of_constraints, x comes before s.
        def custom_sort_x(s):
            if "x" in s:
                return 0
            elif "s" in s:
                return 1
            else:
                return 2

        names_of_variables = sorted(names_of_variables, key = custom_sort_x)
        names_of_constraints = sorted(names_of_constraints, key = custom_sort_x)

        z_line = working_matrix[-1, :]                  # Last line of working_matrix, which contains the values of the function
        z_line_coeffs = z_line[:-1]                     # Coefficients of variables of the function
        z_line_value = z_line[-1]                       # Value of the function
```

```
        values_of_variables = working_matrix[:-1, -1]                    # Values of base variables

        # Find the max (min) of the function
        while True:
            # Check if the solution it is admissible
            if any(values_of_variables < 0):
                if initial_goal == "max":
                    result = pd.DataFrame({
                        f"{initial_goal} value of the function": [z_line_value],
                        "Names of base variables": "There are not admissible solution",
                        "Values of base variables": "There are not admissible solution"
                    })
                    break
                else:
                    if initial_goal == "min":                    # Due to Weak Duality Theorem
                        result = pd.DataFrame({
                            f"{initial_goal} value of the function": [z_line_value],
                            "Names of base variables": "Unbounded Optimal Solution",
                            "Values of base variables": "Unbounded Optimal Solution"
                        })
                        break
                    else:
                        raise Exception("ERROR: Please pass a valid value for the parameter goal ('max' or 'min')")

            # Find a new vertex with a bigger (smaller) value of the function
            constraints_index, names_of_constraints = entryANDexit_criteria(working_matrix, constraints_index, names_of_variables, names_of_constraints)

            if constraints_index == "Unbounded Optimal Solution":
                if initial_goal == "max":
                    result = pd.DataFrame({
                        f"{initial_goal} value of the function": [z_line_value],
                        "Names of base variables": "Unbounded Optimal Solution",
                        "Values of base variables": "Unbounded Optimal Solution"
                    })
                    break
                else:
```

```
                if initial_goal == "min":                                  # Due to Weak Duality Theorem
                    result = pd.DataFrame({
                        f"{initial_goal} value of the function": [z_line_value],
                        "Names of base variables": "There are not admissible solution",
                        "Values of base variables": "There are not admissible solution"
                    })
                    break


        working_matrix = MoveToAnotherVertex(working_matrix, constraints_index)

        z_line = working_matrix[-1, :]                                     # Last line of working_matrix, which contains the values of the function
        z_line_coeffs = z_line[:-1]                                        # Coefficients of variables of the function
        z_line_value = z_line[-1]                                          # Value of the function
        values_of_variables = working_matrix[:-1, -1]                      # Values of base variables

        # Choose if we are in the maximum (minimum) point or not
        continueORNOTcontinue = list()
        for i in range(len(z_line_coeffs)):
            if i in constraints_index:
                continue
            else:
                if round(z_line_coeffs[i], 5) < 0 and goal == "max":
                    continueORNOTcontinue.append(1)
                else:
                    if round(z_line_coeffs[i], 5) > 0 and goal == "max":
                        continueORNOTcontinue.append(0)
                    else:
                        if round(z_line_coeffs[i], 5) == 0:
                            continueORNOTcontinue.append("inf")
                        else:
                            if goal == "min":
                                raise Exception("Something went wrong. This function solves minimization problems using the Duality Theorem.")
```

```
            # Infinite solution
            if any(elemement == 'inf' for elemement in continueORNOTcontinue):
                if initial_goal == "max":
                    result = pd.DataFrame({
                        f"{initial_goal} value of the function": [z_line_value],
                        "Names of base variables": "There are infinite␣
↪solution",
                        "Values of base variables": "There are infinite␣
↪solution"
                    })
                    break
                else:
                    if initial_goal == "min":                                  ␣
↪                           # Due to Weak Duality Theorem
                        result = pd.DataFrame({
                            f"{initial_goal} value of the function":␣
↪[z_line_value],
                            "Names of base variables": "There are not␣
↪admissible solution",
                            "Values of base variables": "There are not␣
↪admissible solution"
                        })
                        break
                    else:
                        raise Exception("ERROR: Please pass a valid value for␣
↪the parameter goal ('max' or 'min')")
            else:
                # Finite solution
                if initial_goal == "max":
                    if sum(continueORNOTcontinue) == 0:
                        result = pd.DataFrame({
                            f"{initial_goal} value of the function":␣
↪[z_line_value],
                            "Names of base variables": [names_of_constraints],
                            "Values of base variables": [values_of_variables]
                        })
                        break
                else:
                    if initial_goal == "min":
                        if sum(continueORNOTcontinue) == 0:
                            names_of_variables_in_the_dual_solution =␣
↪names_of_constraints
                            values_of_variables_in_the_dual_solution =␣
↪values_of_variables
```

```
                           values_of_variables_dual2primal,␣
↪names_of_constraints_dual2primal = Solution_Dual2PrimalProblem(constraints,␣
↪names_of_variables, names_of_variables_in_the_dual_solution,␣
↪values_of_variables_in_the_dual_solution)

                           result = pd.DataFrame({
                               f"{initial_goal} value of the function":␣
↪[z_line_value],
                               "Names of base variables":␣
↪[names_of_constraints_dual2primal],
                               "Values of base variables":␣
↪[values_of_variables_dual2primal]
                           })
                           break
                   else:
                       raise Exception("ERROR: Please pass a valid value for␣
↪the parameter goal ('max' or 'min')")

       result = result.T
       return result
   except:
       raise Exception("ERROR: Something went wrong in the SimplexAlgorithm.␣
↪Please check the passed values.")
```

1.3.2 I/O Knapsack Problem using Branch and Bound Algotithm

The following function solves the famous Knapsack Problem using the Branch and Bound Algorithm. It takes the following inputs:
- *weights* (as a list), representing the space required by each object i;
- *utility* (as a list), representing the importance of each object i;
- *capacity* (as an integer), representing the total space available in the knapsack;

The function finds the combination of objects that maximize the utility, based on the fundamental hypothesis that utility is additive. Specifically, the function returns a Pandas DataFrame that includes which objects are included in the knapsack, the total utility, and the amount of used capacity.

The *KnapsackProblem* function calls two other functions:
- *find_the_branches*: given the capacity, a Pandas DataFrame with initial weights and utility values, and a constraint on weights (e.g., if during the previous branch we chose to include object 2, the constraint would be represented as x1xxx..xxx; otherwise, it would be x0xxx…xxx), this function calculates the maximum total utility, modifies the constraints and weights, and indicates whether a node can be further branched (ok_na = 0) or not (ok_na = 1). All of this information are returned in a Pandas DataFrame;
- *get_my_index*: a simple function that finds where in a DataFrame the values in column 1 meet condition 1 and the values in column 2 meet condition 2.

Note: Although this function is based on the knapsack problem, it can easily be adapted for other problems, such as deciding whether to include or exclude a security in a portfolio (under the assumption that we cannot choose to partially include a security in our portfolio) with the goal of maximizing the portfolio value.

Example
Give the problem:

Objects	*A*	*B*	*C*	*D*	*E*
Weights	25	40	30	50	50
Utility	28	48	37	62	59
Capacity	130				

The *KnapsackProblem* function takes as input:

```
weights = [25, 40, 30, 50, 50]
utility = [28, 48, 37, 62, 59]
capacity = 130
```

And the output will be:

```
             Weights              Obj  Utility  Used capacity (%)
0  [1, 1, 0.0, 1, 0]  [4, 3, 2, 5, 1]    158.0              100.0
```

```
[ ]: def find_the_branches(constraints, df, capacity):
         """
         Given the capacity, a Pandas DataFrame with initial weights and utility␣
      ↪values, and a constraint on weights (e.g., if during the previous branch we␣
      ↪chose to include object 2, the constraint would be represented as x1xxx..xxx;
      ↪ otherwise, it would be x0xxx...xxx), this function calculates the maximum␣
      ↪total utility, modifies the constraints and weights, and indicates whether a␣
      ↪node can be further branched (ok_na = 0) or not (ok_na = 1). All of this␣
      ↪information is returned in a Pandas DataFrame.
         """
         try:
             # Setting the variables
             branch_sum = 0
             branch_weights = [0] * len(df.loc["Weights", ])
             total_utility = 0

             # Initial sum, consider the constraints
             for j_indices1 in range(len(constraints)):
                 if constraints[j_indices1] == 1:
                     branch_sum = branch_sum + df.loc["Weights", ][j_indices1]
                     branch_weights[j_indices1] = 1

             # Check if the initial sum is less than capacity, otherwise stop the␣
      ↪code and return "NA"
             if branch_sum > capacity:
                 return "NA"
             else:
```

```
            for j_branches in range(len(df.loc["Weights", ])):
                # Skip if there're a 0 or a 1 (I've yet consider if there's a 1)
                if (constraints[j_branches] == 0) or (constraints[j_branches]␣
↪== 1):
                    continue
                else:
                    branch_sum = branch_sum + df.loc["Weights", ][j_branches]
                    branch_weights[j_branches] = 1

                    if branch_sum > capacity:
                        branch_sum = branch_sum - df.loc["Weights",␣
↪][j_branches]
                        branch_weights[j_branches] = ((capacity) - branch_sum)/
↪df.loc["Weights", ][j_branches]
                        branch_sum = branch_sum + ((capacity) - branch_sum)
                        break

        for j_utility in range(len(branch_weights)):
            total_utility = total_utility + (df.loc["Utility", ][j_utility] *␣
↪branch_weights[j_utility])

        # Check if the solution is assertable or not
        if sum(branch_weights) % 1 == 0:
            ok_na = 1
        else:
            ok_na = 0

        result = pd.DataFrame({
            "Constraint": [constraints],
            "Weights": [branch_weights],
            "Utility": total_utility,
            "Used capacity (%)": (branch_sum/capacity)*100,
            "OK": ok_na
        }).T

        return result
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

[ ]:
```
def get_my_index(df, condition1, col_name_condition1, condition2,␣
↪col_name_condition2):
    """
    A simple function that finds where in a DataFrame the values in column 1␣
↪meet condition 1 and the values in column 2 meet condition 2.
```

```
    NB: this functions works only with equality conditions (i.e.␣
↪df[col_name_condition1][i] == condition1 works, but␣
↪df[col_name_condition1][i] < condition1 does not work).
    """
    try:
        for i in range(len(df[col_name_condition1])):
            if (df[col_name_condition1][i] == condition1) and␣
↪(df[col_name_condition2][i] == condition2):
                return i
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

[ ]:
```
def KnapsackProblem(weights, utility, capacity):
    """
    The following function solves the famous Knapsack Problem using the Branch␣
↪and Bound Algorithm. It takes the following inputs:
    - weights (as a list), representing the space required by each object *i*; ␣
↪
    - utility (as a list), representing the importance of each object *i*;     ␣
↪
    - capacity (as an integer), representing the total space available in the␣
↪knapsack;

    The function finds the combination of objects that maximise the utility,␣
↪based on the fundamental hypothesis that utility is additive. Specifically,␣
↪the function returns a Pandas DataFrame that includes which objects are␣
↪included in the knapsack, the total utility, and the amount of used capacity.
    """
    try:
        # PREMILIMARY OPERATIONS
        # Creating the DataFrame and ordering objects by u/w ratio.
        uw_ratio = [utility[i]/weights[i] for i in range(len(weights))]        ␣
↪                                    # Utility per weight ratio

        obj = range(1, len(weights) + 1)                                       ␣
↪                                    # "Names" of objects

        df = pd.DataFrame(columns = ['Obj', 'Weights', 'Utility', 'U/W ratio'])

        for i_constraint in range(len(weights)):
            tmp = pd.DataFrame({
                'Obj': [obj[i_constraint]],
                'Weights': [weights[i_constraint]],
                'Utility': [utility[i_constraint]],
                'U/W ratio': [uw_ratio[i_constraint]]
```

```
        })

        df = pd.concat([df, tmp], ignore_index = True)

    df  = df.T                                                                         ␣
↪                                     # Just for My Convenience
    df_orderly = df.sort_values(by = "U/W ratio", axis = 1, ascending =␣
↪False, ignore_index = True)            # Ordering objects by u/w ratio.


    # UPPER BOUND
    # Finding the upper bound
    constraint = ["x"]*len(weights)                                                    ␣
↪                                      # Initial constraint ("x" means that␣
↪there are not constaints)

    upperbound = find_the_branches(constraint, df_orderly, capacity)
    upperbound_weights = (upperbound.loc["Weights", ])[0]
    utility = upperbound.loc["Utility", ]

    # Creating a dataframe where I'll save the results
    branches = pd.DataFrame(columns = ["Constraint", "Weights", "Utility",␣
↪"OK"])
    # Saving the values of the upper bound
    upperbound_df = pd.DataFrame({
        "Constraint": [constraint],
        "Weights": [upperbound_weights],
        "Utility": utility,
        "OK": upperbound.loc["OK"],
        "Used capacity (%)": upperbound.loc["Used capacity (%)"],
        "BranchOff": "No"
    })
    branches = pd.concat([branches, upperbound_df], ignore_index = True)


    # BRANCHES
    options = [0, 1]                                                                   ␣
↪                                      # Options for the constraints: each␣
↪times I've to check if the decimal number could be 0 or 1
    counter_cylces = 0                                                                 ␣
↪                                      # If the cycles will be more than 10^5,␣
↪I'll stop the code and return the default result
    result = "There is not a integer solution"                                         ␣
↪                                      # Defual result

    while True:
```

```
        # Get the number of node with highest utility and that is not␣
↪branch off yet
        max_utility = branches.query("BranchOff == 'No'")['Utility'].max()
        index_of_branch = get_my_index(branches, max_utility, "Utility",␣
↪"No", "BranchOff")

        weights_tmp = branches["Weights"][index_of_branch]
        constraint_tmp = branches["Constraint"][index_of_branch]

        if sum(weights_tmp) % 1 == 0:                                          ␣
↪                            # If the sum of final_weigths has no␣
↪remainder means that we've find the optimal solution
            break
        else:
            for i_constraint in range(len(weights_tmp)):
                if not((weights_tmp[i_constraint] == 1) or␣
↪(weights_tmp[i_constraint] == 0)):                    # If the value in the␣
↪position i is not 1 or 0 means that we've find where branch off
                    for i_options in range(len(options)):
                        new_constraint_tmp = constraint_tmp.copy()
                        new_constraint_tmp[i_constraint] =␣
↪options[i_options]
                        branch_tmp = find_the_branches(new_constraint_tmp,␣
↪df_orderly, capacity)

                        if type(branch_tmp) == str:                            ␣
↪                            # If find_the_branches returns "NA" (a␣
↪string) means that there is not a branch available
                            continue

                        newrow = pd.DataFrame({
                            "Constraint": [new_constraint_tmp],
                            "Weights": [(branch_tmp.loc["Weights", ])[0]],
                            "Utility": branch_tmp.loc["Utility", ],
                            "OK": branch_tmp.loc["OK"],
                            "Used capacity (%)": branch_tmp.loc["Used␣
↪capacity (%)"],
                            "BranchOff": "No"
                        })
                        branches = pd.concat([branches, newrow],␣
↪ignore_index = True)

                    branches["BranchOff"][index_of_branch] = "Yes"             ␣
↪                            # Say that this node is branched off

        # If cycles are more than 10^5, stop the code
```

```
            counter_cylces +=1
            if(counter_cylces> 10*5):
                return result

            # Get the result
            if any(branches["OK"] == 1):
                # Branches admissible and not admissible
                branches_ok = branches[branches['OK'] == 1]
                branches_ok = branches_ok.reset_index(drop = True)
                branches_na = branches[(branches['OK'] == 0) &␣
↪(branches['BranchOff'] == 'No')]

                max_utility_OK = branches_ok["Utility"].max()
                max_utility_NA = branches_na["Utility"].max()

                tmp_result = list(branches_ok["Utility"])
                index_of_result = tmp_result.index(max_utility_OK)

                result = pd.DataFrame({
                    "Weights": [branches_ok["Weights"][index_of_result]],
                    "Obj": [list(df_orderly.loc["Obj",:].astype(int))],
                    "Utility": [branches_ok["Utility"][index_of_result]],
                    "Used capacity (%)": [branches_ok["Used capacity␣
↪(%)"][index_of_result]],
                })

                if float(max_utility_OK) >= float(max_utility_NA):
                    break
                else:
                    continue

        return result
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.4 Dynamic Models

- Consensus Dynamic Model
- Linear Dynamic Population Redistribution Model

Here I report the theory on which the two models are based.

Consensus Dynamic Model
The Consensus Dynamic Model (in a popolutaion of $n \geq 2$ individuals) is based on the following equation:

$$x_t = Cx_{t-1} \quad \forall t > 0$$

where x_t is a vector which represents the opinion of individuals.
The transition matrix C determine the time-invariant weighting mechanism that describes the revision of individual opinions in each iteration of the model. It is assumed that the transition matrix C, a square matrix of order n, is positive, simple (there exists a basis of eigenvectors), and row-stochastic.

Each element of the transition matrix $c_{ij} \in (0, 1)$ represents the influential weight that individual i assigns to the opinion of individual j when revising and updating their own opinion, with $i, j = 1, \dots, n$. In this model, the transition matrix C is constructed based on the interaction matrix $V = [v_{ij}]$ and the vector $u = (u_1, \dots, u_n)$ of individual propensities $u_i \in (0, 1)$ to revise their opinions, with $i = 1, \dots, n$. The diagonal of the interaction matrix is zero, and each off-diagonal element $v_{ij} \in (0, 1)$ with $i \neq j$ represents the degree of authority that individual i attributes to individual j.

The solution path of the dynamic model involves, in addition to the construction of the transition matrix C, the determination of the asymptotic solution indicated by the vector $x(t = \infty)$, which represents the convergence of the linear dynamics toward a stable profile of the opinions of the n individuals.

Considering that the dominant right eigenvector of the transition matrix is $r = (1/n, \dots, 1/n)$, the only stable solutions are those consensual ones, where all opinions are aligned to the same consensual opinion. Denoting the invariant scalar of the dynamic model by $\tilde{x} = s^T x$, where s is the dominant left eigenvector of the transition matrix, and knowing that the asymptotic solution is a multiple of the dominant right eigenvector r, we can use the fact that $s^T r = 1/n$ to conclude that the linear dynamics asymptotically converge to the stable profile $x(t = \infty) = n\tilde{x}r = (\tilde{x}, \dots, \tilde{x})$, where the invariant scalar $\tilde{x} = s^T x$ corresponds to the asymptotic consensual opinion.

Finally, since the asymptotic consensual opinion can be calculated by directly weighting the initial opinions with the components of the dominant left eigenvector, $\tilde{x} = s^T x(t = 0)$, the components of s indicate the relative importance that each of the n individuals had in determining the value of the consensual opinion.

Linear Dynamic Population Redistribution Model
Consider the linear dynamic model of redistribution of a population of $N \geq 2$ individuals over $n \geq 2$ types of residential areas: for example, with $n = 3$, the areas are urban ($i = 1$), suburban ($i = 2$), and rural ($i = 3$). The distribution of individuals among the n residential areas is indicated by the vector

$$x(t) = (x_1(t), \dots, x_n(t)) \quad \forall t \geq 0$$

where $x_i(t)$ denotes the number of individuals residing in area $i = 1, \dots, n$ at time $t = 0, 1, 2, \dots$, always maintaining

$$x_1(t) + \dots + x_n(t) = N.$$

The linear redistribution dynamics for residential areas, expressed by the iterative law

$$x(t) = Cx(t - 1) \quad \forall t \geq 0$$

aims to represent the change in the population distribution across the n residential areas due to the annual transfer of individuals from one residential area to another, starting from an initial distribution $x(t = 0)$.

The transition matrix $C = [c_{ij}]$ of the linear dynamics describes the time-invariant pattern of these annual transfers of individuals from one residential area to another. It is assumed that the transition matrix C, a square matrix of order n, is positive, simple (there exists a basis of eigenvectors), and column-stochastic. Each element of the transition matrix, $c_{ij} \in (0, 1)$, represents the fraction of residents in area j that transfer each year to area i, with $i, j = 1, \ldots, n$.

[ ]:
```
def DynamicModels(matrix, x_t0, x_tn, type_of_model, u = None, N = None):
    """
    This function resolves two types of Dynamic Models seen in class: the␣
 ↪Consensus Dynamics Model and the Linear Dynamic Population Redistribution␣
 ↪Model. It takes as input:
    - a matrix (np.array) representing V in the case of the Consensus Dynamics␣
 ↪Model or C in the case of the Linear Dynamic Population Redistribution Model;
    - a vector (list) representing x(t0);
    - an integer value for t representing the time step for the finite solution␣
 ↪(e.g., if we want to compute x(t10), then t will be 10);
    - type_of_model: "cd" for the Consensus Dynamics Model or "pop" for the␣
 ↪Linear Dynamic Population Redistribution Model;
    - a vector (list) representing u in the case of the Consensus Dynamics␣
 ↪Model, or an integer N in the case of the Linear Dynamic Population␣
 ↪Redistribution Model.
    """
    try:
        result = pd.DataFrame(columns = ["Asymptotic solution", f"x_t{x_tn}",␣
 ↪"Dominant eigenvector"])

        # Linear Dynamic Population Redistribution Model
        if type_of_model == "pop":
            # Computing and sorting eigenvalues, eigenvectors
            eigenvalues, eigenvectors = np.linalg.eig(matrix) # (right␣
 ↪eigenvectors)
            indices = np.argsort(eigenvalues)[::-1]
            eigenvalues = eigenvalues[indices]
            eigenvectors = eigenvectors[:, indices]

            if round(eigenvalues[0], 5) == 1:
                # Asymptotic solution
                r = eigenvectors[:,0] / sum(eigenvectors[:,0])
                asymptotic_solution = N * r

                # Solution at t_n
                x_tminus1 = x_t0
                for i in range(x_tn):
```

```
                x_t = np.matmul(matrix, x_tminus1)
                x_tminus1 = x_t

            # Return the initial matrix
            return_matrix = matrix
            dominant_eigenvector = r
        else:
            print("ERROR: The matrix you provided does not have an␣
↪eigenvalue equal to 1.")
    else:
        # Consensus Dynamics Model
        if type_of_model == "cd":
            # Transition matrix C
            transition_matrix = np.empty((len(matrix[:,0]), len(matrix[0])))
            for i in range(len(matrix[:,0])):
                for j in range(len(matrix[0])):
                    if i == j:
                        transition_matrix[i, j] = 1 - u[i]
                    else:
                        transition_matrix[i, j] = u[i] * (matrix[i, j]/
↪sum(matrix[i]))

            # Asymptotic solution
            eigenvectors = eigeinvectors_calculator(transition_matrix,␣
↪"left")
            # Get the index where eigenvalue equals to 1
            eigenvalues = eigenvectors["Eigenvalue"]
            for z in range(len(eigenvalues)):
                if math.isclose(eigenvalues[z][0], 1):
                    index_eigenvalues = z
                    break
            # Get the eigenvector which corresponds to eigenvalue 1
            s = list()
            for i in range(eigenvectors["Eigenvector"][0][0].shape[0]):
                s.append(eigenvectors.iloc[index_eigenvalues, 1][0][i][0])
            s = s / sum(s)

            asymptotic_solution = [round(np.dot(s, x_t0), 5)] * len(x_t0)
            dominant_eigenvector = s

            # Solution at t_n
            x_tminus1 = x_t0
            for i in range(x_tn):
                x_t = np.matmul(transition_matrix, x_tminus1)
                x_tminus1 = x_t

            # Return the transition matrix
```

```
                return_matrix = transition_matrix

        # Result
        dominant_eigenvector = [round(num, 5) for num in dominant_eigenvector]
        newrow = {"Asymptotic solution" : np.around(asymptotic_solution, 5),␣
↪f"x_t{x_tn}": np.around(x_t, 5), "Dominant eigenvector" :␣
↪dominant_eigenvector}
        result.loc[len(result)] = newrow

        return result, return_matrix
    except:
        raise Exception("ERROR: Something went wrong. Please pass valid values␣
↪for the parameters")
```

1.5 Test

1.5.1 Introductory matrix algebra

[ ]:
```
# Addition of two matrices
m1 = np.array([[1,2,3],[4,5,6]])
m2 = np.array ([[0,-1,2], [-2,1,0]])
assert np.array_equal(matrixplusmatrix(m1,m2), (m1 + m2))
print("The MatrixPlusMatrix function works correctly")

# Multiplication of two matrices
m3 = np.array([[2, -1], [0,1]])
m4 = np.array([[1,2,3], [3,1,0]])
assert np.array_equal(matrixtimesmatrix(m3,m4), np.dot(m3,m4))
print("The MatrixTimesMatrix function works correctly")

# Determinant of a matrix
m5 = np.array([[0]])
m6 = np.array([[2,4], [1,2]])
m7 = np.array ([[1,-1,-4], [3,1,-1], [5,3,2]])
m8 = np.array([[2,6,-2,2], [-2,4,0,3], [-3,1,1,2], [1,3,-1,2]])
assert determinant(m5) == round(np.linalg.det(m5),2) #1x1
assert determinant(m6) == round(np.linalg.det(m6),2) #2x2
assert determinant(m7) == round(np.linalg.det(m7),2) #Sarrus
assert determinant(m8) == round(np.linalg.det(m8),2) #4x4
print("The Determinant function works correctly")

# Inverse of a matrix
m9 = np.array([[1,3,4], [1,1,4], [-1,0,1]])
assert np.allclose(inversematrix(m9), np.linalg.inv(m9))
print("The InverseMatrix function works correctly")

# Minor of a matrix
```

```python
m10 = np.array([[1,0,-1,2],[-1,1,-2,0],[-2,-1,1,2]])
assert len(minor(m10)) == 34
assert len(minor(m10, order = 1)) == 12
assert len(minor(m10, order = 2)) == 18
assert len(minor(m10, order = 3)) == 4
print("The Minor function works correctly")

# Rank of a matrix
m11 = np.array([[1,-2,0],[0,1,1],[-1,3,1]])
m12 = np.array([[2,6,-2,2], [-2,4,0,3], [-3,1,1,2], [1,3,-1,1]])
m13 = np.array([[1,0,-1,2],[-1,1,-2,0],[2,-1,1,2]])
assert rank(m11) == np.linalg.matrix_rank(m11)
assert rank(m12) == np.linalg.matrix_rank(m12)
assert rank(m13) == np.linalg.matrix_rank(m13)
print("The Rank function works correctly")

# Gaussian elimination method for solving a linear system
# Check diagonal
assert check_diagonal(np.array([[1, 1, 1], [0, 1, 2], [0, 0, 2]])) == True
assert check_diagonal(np.array([[1, 1, 1], [0, 1, 2], [0, 1, 2]])) == False
assert check_diagonal(np.array([[0, 1, 1], [0, 1, 2], [0, 1, 2]])) == False
assert check_diagonal(np.array([[1, 1, 1], [0, 0, 2], [0, 1, 2]])) == False
assert check_diagonal(np.array([[1, 1, 1], [0, 1, 2], [0, 0, 0]])) == True
print("The CheckDiagonal function works correctly")
# Gaussian elimination method for solving a linear system
m14 = np.array([[1,-1,4],[3,1,-1],[5,3,2]])
m15 = np.array([[0],[3],[6]])
assert np.array_equal(guassian_elimination(m14, m15), np.array([[1, -1, 4, 0],␣
↪[0, 1, -3.25, 0.75], [0, 0, 0, 0]]))
print("The GuassianElimination function works correctly")
# Solve a linear system
assert solve_linearsystem(m14, m15) == ["x1 = ['-1.0x2' '4.0x3' 0.0]", "x2 =␣
↪['-3.25x3' 0.75]"]
print("The SolveLinearSystem function works correctly")

# Eigenvectors
matrix = np.array([[1, -2, 2],
                   [-1, 2, 1],
                   [0, 0, -1]])

eigeinvectors_right = eigeinvectors_calculator(matrix, "right")
alpha1r = 1/(eigeinvectors_right["Eigenvector"][0][0][0] * 0.5)
alpha2r = -1/(eigeinvectors_right["Eigenvector"][1][0][0])
alpha3r = -1/(eigeinvectors_right["Eigenvector"][2][0][0] * 0.5)
assert np.allclose((eigeinvectors_right["Eigenvector"][0])[0]*alpha1r, np.
↪array([[2], [1], [0]]), atol=0.01, rtol=0.01) == True
```

```
assert np.allclose((eigeinvectors_right["Eigenvector"][1])[0]*alpha2r, np.
↪array([[-1], [1], [0]]), atol=0.01, rtol=0.01) == True
assert np.allclose((eigeinvectors_right["Eigenvector"][2])[0]*alpha3r, np.
↪array([[-2], [-1], [1]]), atol=0.01, rtol=0.01) == True

eigeinvectors_left = eigeinvectors_calculator(matrix, "left")
alpha1l = 1/(eigeinvectors_left["Eigenvector"][0][0][0])
alpha2l = -1/(eigeinvectors_left["Eigenvector"][1][0][0])
assert np.allclose((eigeinvectors_left["Eigenvector"][0])[0]*alpha1l, np.
↪array([[1], [1], [3]]), atol=0.01, rtol=0.01) == True
assert np.allclose((eigeinvectors_left["Eigenvector"][1])[0]*alpha2l, np.
↪array([[-1], [2], [0]]), atol=0.01, rtol=0.01) == True
assert np.allclose((eigeinvectors_left["Eigenvector"][2])[0], np.array([[0],␣
↪[0], [1]]), atol=0.01, rtol=0.01) == True

print("The eigeinvectors_calculator function works correctly")
```

```
The MatrixPlusMatrix function works correctly
The MatrixTimesMatrix function works correctly
The Determinant function works correctly
The InverseMatrix function works correctly
The Minor function works correctly
The Rank function works correctly
The CheckDiagonal function works correctly
The GuassianElimination function works correctly
The SolveLinearSystem function works correctly
The eigeinvectors_calculator function works correctly
```

1.5.2 Linear programming

[ ]:
```
# Simplex Algorithm
# MAX
function_to_minimax = np.array([[6, 2, 4, 0]])
constraints = np.array([[1, 2, 3, 60],
                        [2, 1, 1, 30]])
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[0,0] ==␣
↪108
assert np.array_equal(SimplexAlgorithm(function_to_minimax, constraints, "max").
↪iloc[2,0], [6, 18])

function_to_minimax = np.array([[-1, 2, 0]])
constraints = np.array([[-1, 2, 10],
                        [1, 1, 11],
                        [4, -7, 0]])
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[0,0] == 10
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[2,0] ==␣
↪"There are infinite solution"
```

```python
function_to_minimax = np.array([[6, 4, 15, 0]])
constraints = np.array([[3, 1, 5, 4],
                        [1, 1, 3, 5]])
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[0,0] == 16
assert np.array_equal(SimplexAlgorithm(function_to_minimax, constraints, "max").
 ↪iloc[2,0], [4, 1])

# Ottimo illimitato (vedi appunti)

function_to_minimax = np.array([[1, 1, 0]])
constraints = np.array([[-4, 1, 3],
                        [2, -1, 6]])
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[0,0] == 3
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[2,0] ==␣
 ↪"Unbounded Optimal Solution"

function_to_minimax = np.array([[1, -1, 0]])
constraints = np.array([[1, 1, 5],
                        [1, -2, 2],
                        [0, 2, 3]])
assert SimplexAlgorithm(function_to_minimax, constraints, "max").iloc[0,0] == 3
assert np.array_equal(SimplexAlgorithm(function_to_minimax, constraints, "max").
 ↪iloc[2,0], [4, 1, 1])

print("The function SimplexAlgorithm work correctly when goal == 'max'")

# MIN
function_to_minimax = np.array([[8, 4, 1, 0]])
constraints = np.array([[1, -1, 1, 1],
                        [1, 1, 0, 3]])
assert SimplexAlgorithm(function_to_minimax, constraints, "min").iloc[0,0] == 16
assert np.array_equal(SimplexAlgorithm(function_to_minimax, constraints, "min").
 ↪iloc[2,0], [0, 3, 4])

constraints = np.array([[1,1,0,1],
                        [1,-2,1,-1]])
function_to_minimax = np.array([[5, 2, 3, 0]])
assert SimplexAlgorithm(function_to_minimax, constraints, "min").iloc[0,0] == 3
assert np.array_equal(SimplexAlgorithm(function_to_minimax, constraints, "min").
 ↪iloc[2,0], [0.33333, 0.66667])

constraints = np.array([[2, 3, 15],
                        [1, 1, 6],
                        [5, 3, 9]])
function_to_minimax = np.array([[20, 24, 0]])
```

```
assert SimplexAlgorithm(function_to_minimax, constraints, "min").iloc[0,0] ==␣
↪132
assert np.array_equal(SimplexAlgorithm(function_to_minimax, constraints, "min").
↪iloc[2,0], [3, 3, 15])
print("The function SimplexAlgorithm work correctly when goal == 'min'")


# Knapsack probelm
assert KnapsackProblem(weights = [36, 24, 30, 32, 26], utility = [54, 18, 60,␣
↪32, 13], capacity = 91).iloc[0, 0] == [1,1,0,1,0]
assert KnapsackProblem(weights = [10, 8, 15, 12, 9], utility = [10, 4, 20, 24,␣
↪15], capacity = 40).iloc[0, 0] == [1,1,1,0,0]
assert KnapsackProblem(weights = [25, 40, 30, 50, 50], utility = [28, 48, 37,␣
↪62, 59], capacity = 130).iloc[0, 0] == [1,1,0,1,0]
assert KnapsackProblem(weights = [23, 12, 10, 30, 13], utility = [50, 25, 15,␣
↪55, 20], capacity = 50).iloc[0, 0] == [1,1,0,1,0]
assert KnapsackProblem(weights = [15, 14, 10, 11, 4], utility = [40, 42, 32,␣
↪25, 9], capacity = 25).iloc[0, 0] == [1,1,0,0,0]
print("The KnapsackProblem function works correctly")
```

```
The function SimplexAlgorithm work correctly when goal == 'max'
The function SimplexAlgorithm work correctly when goal == 'min'
The KnapsackProblem function works correctly
```

1.5.3 Dynamic Models

[ ]:
```
# Dynamic models
# Consensum Dynamics
matrix = np.array([[0, 0.5, 0.25],
                   [0.5, 0, 0.25],
                   [0.25, 0.25, 0]])
assert np.array_equal(DynamicModels(matrix, x_t0 = [4, 8 ,2], x_tn = 1,␣
↪type_of_model = "cd", u = [0.5, 0.5, 0.5])[0].loc[:, "Asymptotic␣
↪solution"][0], [5, 5, 5])

matrix = np.array([[0, 1/2, 1/3],
                   [1/2, 0, 1/4],
                   [1/3, 1/4, 0]])
assert np.array_equal(DynamicModels(matrix, x_t0 = [4, 8 ,2], x_tn = 1,␣
↪type_of_model = "cd", u = [0.5, 0.5, 0.5])[0].loc[:, "Asymptotic␣
↪solution"][0], [4.84614, 4.84614, 4.84614])

matrix = np.array([[0, 0.5, 0.25],
                   [0.5, 0, 0.25],
                   [0.5, 1/3, 0]])
```

```
assert np.array_equal(DynamicModels(matrix, x_t0 = [4, 8 ,2], x_tn = 1,␣
 ↪type_of_model = "cd", u = [0.5, 0.5, 0.5])[0].loc[:, "Asymptotic␣
 ↪solution"][0], [4.94001, 4.94001, 4.94001])

matrix = np.array([[0, 0.5, 1/3],
                   [0.5, 0, 0.25],
                   [0.25, 0.25, 0]])
assert np.array_equal(DynamicModels(matrix, x_t0 = [4, 8 ,2], x_tn = 1,␣
 ↪type_of_model = "cd", u = [0.5, 0.5, 0.5])[0].loc[:, "Asymptotic␣
 ↪solution"][0], [4.89553, 4.89553, 4.89553])

print("The DynamicModels function works correctly for the Consensum Dynamics␣
 ↪Model")

# Linear Dynamic Population Redistribution Model
matrix = np.array([[0.6, 0.1, 0.1],
                   [0.3, 0.8, 0.2],
                   [0.1, 0.1, 0.7]])
assert np.array_equal(DynamicModels(matrix, x_t0 = [40, 40, 20], x_tn = 1,␣
 ↪type_of_model = "pop", N = 100)[0].loc[:, "Asymptotic solution"][0], [20,␣
 ↪55, 25])

matrix = np.array([[0.6, 0.1, 0.1],
                   [0.3, 0.8, 0.3],
                   [0.1, 0.1, 0.6]])
assert np.array_equal(DynamicModels(matrix, x_t0 = [40, 40, 20], x_tn = 1,␣
 ↪type_of_model = "pop", N = 100)[0].loc[:, "Asymptotic solution"][0], [20,␣
 ↪60, 20])

matrix = np.array([[0.7, 0.1, 0.1],
                   [0.2, 0.8, 0.3],
                   [0.1, 0.1, 0.6]])
assert np.array_equal(DynamicModels(matrix, x_t0 = [40, 40, 20], x_tn = 1,␣
 ↪type_of_model = "pop", N = 100)[0].loc[:, "Asymptotic solution"][0], [25,␣
 ↪55, 20])

matrix = np.array([[0.8, 0.1, 0.1],
                   [0.1, 0.7, 0.3],
                   [0.1, 0.2, 0.6]])

assert np.array_equal(DynamicModels(matrix, x_t0 = [40, 40, 20], x_tn = 1,␣
 ↪type_of_model = "pop", N = 100)[0].loc[:, "Asymptotic solution"][0], [33.
 ↪33333, 38.88889, 27.77778])

print("The DynamicModels function works correctly for the Linear Dynamic␣
 ↪Population Redistribution Model")
```

```
The DynamicModels function works correctly for the Consensum Dynamics Model
The DynamicModels function works correctly for the Linear Dynamic Population
Redistribution Model
```