Numerical Optimisation. Project 1

Team Information

Group 1
Participants information in alphabetical order

#	Name	Lastname	Matr Number
1	David	Kürnsteiner	11820336
2	Christian	Peinthor	11815592
3	Elias	Ramoser	11918558
4	Georg	Storz	11918811

Implementation

Imports

Describe how to install additional packages, if you have some, here

In [1]:
```
import numpy as np
import scipy
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from numpy.linalg import *
from sklearn.datasets import make_spd_matrix
```

Stopping criteria

Place for additional comments and argumentation

Simplex Method

In [2]:

```
def simplex_method(x, A, b):
    t = [a + [b[ind]] for ind, a in enumerate(A)]
    z = x + [0]
    table = t + [z]

    while can_be_improved(table):
        pivot_row, pivot_col = get_pivot_position(table)
        updated_table = [[] for y in table]
        pivot_val = table[pivot_row][pivot_col]
        updated_table[pivot_row] = np.array(table[pivot_row]) / pivot_val

        for row_index, row in enumerate(table):
            if row_index != pivot_row:
                multiplier = np.array(updated_table[pivot_row]) * table[row_index][p
                updated_table[row_index] = np.array(table[row_index]) - multiplier

        table = updated_table

    solutions = []
    cols = np.array(table).T
    for column in cols[:-1]:
        if sum(column) == 1 and len([c for c in column if c == 0]) == len(column) -
            one_in_list = list(column).index(1)
            solutions.append(cols[-1][one_in_list])
        else:
            solutions.append(0)

    return solutions
```

In [3]:

```
def can_be_improved(table):
    z = table[-1]
    improveable = False
    for elem in z[:-1]:
        if elem > 0:
            improveable = True
    return improveable
```

In [4]:

```
def get_pivot_position(table):
    z = table[-1]
    column_index = next(index for index, elem in enumerate(z[:-1]) if elem > 0)

    restrictions = []
    for row in table[:-1]:
        elem_ = row[column_index]
        if elem_ <= 0:
            restrictions.append(float('Inf'))

        else:
            restrictions.append(row[-1] / elem_)

    num_of_inf_res = 0
    for res in restrictions:
        if res == float('Inf'):
            num_of_inf_res += 1

    if num_of_inf_res == len(restrictions):
        raise Exception('This linear program is unbounded!')

    min_val = min(restrictions)
```

```python
    row_index = restrictions.index(min_val)

    return row_index, column_index
```

Varibales scaling

Place your reasoning here, how your algorithm behave with respect to this problem. You can also try rescaling your problems This is additional task, which can earn you several points.

Stabilising algorithm

Place your reasoning here, how your algorithm behave with respect to this problem. You can also try rescaling your problems This is additional task, which can earn you several points.

In [5]:
```python
#your function for stabilising goes here
```

Fighting floating-point numbers and roundoff error

Place your reasoning, how your algorithm behave with respect to this problem. You can also try rescaling your problems This is additional task, which can earn you several points.

Inverting matrices

Place for additional comments and argumentation

In [6]:
```python
#your function for invertion goes here
```

Gradients calculation

Place for additional comments and argumentation

In [7]:
```python
#your function for gradient approximation goes here
```

Additional objects you implemented

Place for additional comments and argumentation

In [8]:
```
#your code goes here
```

Optimising algorithm itself

Place for additional comments and argumentation

In [9]:
```
#your code goes here
```

Testing on 5-10 variables, Quadratic objective

Implement functions to optimise over

Place for additional comments and argumentation

In [10]:
```
#your code goes here
```

Run 5 tests

Note: *After every test print out the resulsts.*
For your convinience we implemented a function which will do it for you. Function can be used in case after running optimisation you return $x_{optimal}$, and if you have implemented your gradient approximation. Feel free to bring your adjustments.
Additionaly print how many iterations your algotithm needed. You might also provide charts of your taste (if you want).

Place for your additional comments and argumentation

In [11]:
```
def final_printout(x_0,x_optimal,x_appr,f,grad,args,tolerance):
    """
    Parameters
    --------------------------------------------------------------------------------
    x_0: numpy 1D array, corresponds to initial point
```

```
    x_optimal: numpy 1D array, corresponds to optimal point, which you know, or have
    x_appr: numpy 1D array, corresponds to approximated point, which your algorithm
    --------------------------------------------------------------------------------
    f: function which takes 2 inputs: x (initial, optimal, or approximated)
                                      **args
       Function f returns a scalar output.
    --------------------------------------------------------------------------------
    grad: function which takes 3 inputs: x (initial, optimal, or approximated),
                                         function f,
                                         args (which are submitted, because you migh
                                              to call f(x,**args) inside your gradie
          Function grad approximates gradient at given point and returns a 1d np arr
    --------------------------------------------------------------------------------
    args: dictionary, additional (except of x) arguments to function f
    tolerance: float number, absolute tolerance, precision to which, you compare opt
    """

    print(f'Initial x is :\t\t{x_0}')
    print(f'Optimal x is :\t\t{x_optimal}')
    print(f'Approximated x is :\t{x_appr}')
    print(f'Is close verificaion: \t{np.isclose(x_appr,x_optimal,atol=tolerance)}\n'
    f_opt = f(x_optimal,**args)
    f_appr = f(x_appr,**args)
    print(f'Function value in optimal point:\t{f_opt}')
    print(f'Function value in approximated point:   {f_appr}')
    print(f'Is close verificaion:\t{np.isclose(f_opt,f_appr,atol=tolerance)}\n')
    print(f'Gradient approximation in optimal point is:\n{grad(f,x_optimal,args)}\n'
    grad_appr = grad(f,x_appr,args)
    print(f'Gradient approximation in approximated point is:\n{grad_appr}\n')
    print(f'Is close verificaion:\n{np.isclose(grad_appr,np.zeros(grad_appr.shape),a
```

In [12]:

```
#your code goes here

# function: x1 + x2
# subject to
#     -x1 + x2 + x3 = 2
#     x1 + x4 = 4
#     x2 + x5 = 4

c = [1, 1, 0, 0, 0]
A = [
        [-1, 1, 1, 0, 0],
        [ 1, 0, 0, 1, 0],
        [ 0, 1, 0, 0, 1]
    ]
b = [2, 4, 4]

solution = simplex_method(c, A, b)
print(f'Solution by simplex: {solution}')
print(f'Solution by hand: {[4, 4, 2, 0, 0]}')
print()


# function: x1 - x2
# subject to
#     -x1 + x2 + x3 + x4 = 1
#     x1 + x3 = 6

c = [1, -1, 0, 0]
A = [
        [-1, 1, 1, 1],
        [ 1, 0, 1, 0]
    ]
b = [1, 6]
```

```
solution = simplex_method(c, A, b)
print(f'Solution by simplex: {solution}')
print(f'Solution by hand: {[6, 0, 0, 7]}')
print()

# function: -x1 +5x2
# subject to
#     x1 + 2x2 + x3 + x4 = 4
#     x1 + x3 - x4 = 5

c = [-1, 5, 0, 0]
A = [
        [1, 2, 1, 1],
        [ 1, 0, 1, -1]
    ]
b = [4, 5]

solution = simplex_method(c, A, b)
print(f'Solution by simplex: {solution}')
print(f'Solution by hand: {[0, 2, 0, 0]}')
print()
```

```
Solution by simplex: [4.0, 4.0, 2.0, 0, 0]
Solution by hand: [4, 4, 2, 0, 0]

Solution by simplex: [6.0, 0, 0, 7.0]
Solution by hand: [6, 0, 0, 7]

Solution by simplex: [0, 2.0, 0, 0]
Solution by hand: [0, 2, 0, 0]
```

Here is some place for your analysis. How the behavour of algorithm changed after adjustments? What are specific details, differences you noticed with respect to other algorithms behaviour.

Testing on functions of 1-2 variables, Non-quadratic objective

---

Implement functions to optimise over

Place for additional comments and argumentation

In [13]:

```
#your code goes here
```

Run 5 tests

Place for your additional comments and argumentation

In [14]:

```
#your code goes here
```

Here is some place for your analysis. How the behavour of algorithm changed after adjustments? What are specific details, differences you noticed with respect to other algorithms behaviour.

Template for teachers' tests

Set up a template, how one can run your code

Template should include sceletons for:

- *custom function to optimise over*
- *values initialisation to submit into otimising algorithm*
- *optimiser function call*
- *report print out call*

Provide descriptions and comments.

In [15]:

```
# We have to provide the function in tabular form

# c are the goal function parameters
# A are the parameters of the left side variables of the constraints
# b are the restriction values on the right side of the restrictio

# For testing your own function, please just change the values of the example functi


# function: x1 + x2
# subject to
#     -x1 + x2 + x3 = 2
#     x1 + x4 = 4
#     x2 + x5 = 4

c = [1, 1, 0, 0, 0]
A = [
        [-1, 1, 1, 0, 0],
        [ 1, 0, 0, 1, 0],
        [ 0, 1, 0, 0, 1]
    ]
b = [2, 4, 4]
solution_by_hand = [4, 4, 2, 0, 0]

solution = simplex_method(c, A, b)
print(f'Solution by simplex: {solution}')
print(f'Solution by hand: {solution_by_hand}')
print()
```

```
Solution by simplex: [4.0, 4.0, 2.0, 0, 0]
Solution by hand: [4, 4, 2, 0, 0]
```

In [ ]: