exercise

January 8, 2024

1 Exercise Notebook

This is a jupyter (jupyterlab) notebook, a way to easily execute python interactively (with graphical output).

[1]:
```
import numpy as np
```

[2]:
```
# nicely display matrices
from IPython.display import display, Math

def print_matrix(array, prefix=''):
    matrix = ''
    for row in array:
        try:
            for number in row:
                matrix += f'{number}&'
        except TypeError:
            matrix += f'{row}&'
        matrix = matrix[:-1] + r'\\'
    display(Math(prefix + r'\begin{bmatrix}'+matrix+r'\end{bmatrix}'))
```

2 Simple example of simplex iterations

2.0.1 Example LP in matrix form

[3]:
```
A = np.array([[1, -1, 1, 0, 0], [2, -1, 0, 1, 0], [0, 1, 0, 0, 1]], dtype=np.
↪float64)
print_matrix(A, prefix="A=")
b = np.array([[1],[3],[5]], dtype=np.float64)
print_matrix(b, prefix="b=")
c = np.array([[4],[3],[0],[0],[0]], dtype=np.float64)
print_matrix(c, prefix="c=")
```

$$A = \begin{bmatrix}1.0 & -1.0 & 1.0 & 0.0 & 0.0\\2.0 & -1.0 & 0.0 & 1.0 & 0.0\\0.0 & 1.0 & 0.0 & 0.0 & 1.0\end{bmatrix}$$

$$b = \begin{bmatrix} 1.0 \\ 3.0 \\ 5.0 \end{bmatrix}$$

$$c = \begin{bmatrix} 4.0 \\ 3.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

2.0.2 Initial basis

Since this is python, we use zero-based indexing as opposed to one-based indexing (so x_0 is the first variable).

[4]:
```
basic = [2,3,4]  #\mathcal{B}
nonbasic = [0,1] #\mathcal{N}

def get_columns(A, cols):
    return A[:,cols]

B = get_columns(A, basic)
print_matrix(B, prefix="B = ")
N = get_columns(A, nonbasic)
print_matrix(N, prefix="N = ")
```

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$N = \begin{bmatrix} 1.0 & -1.0 \\ 2.0 & -1.0 \\ 0.0 & 1.0 \end{bmatrix}$$

2.0.3 Initial basic solutions

[5]:
```
x_B_s = np.array(b, dtype=np.float64)
print_matrix(x_B_s, prefix=r"x_{\mathcal{B}}^* = b = ")
z_N_s = -np.array(c[:2], dtype=np.float64)
print_matrix(z_N_s, prefix=r"z_{\mathcal{N}}^* = -c_{\mathcal{N}} = ")
```

$$x^*_{\mathcal{B}} = b = \begin{bmatrix} 1.0 \\ 3.0 \\ 5.0 \end{bmatrix}$$

$$z^*_{\mathcal{N}} = -c_{\mathcal{N}} = \begin{bmatrix} -4.0 \\ -3.0 \end{bmatrix}$$

2.1 First two steps: Find entering variable or determine optimality

[6]:
```
def select_entering_min(z_n_star):
    result_value = 0
    result_index = None
    for i, z_i_star in enumerate(z_n_star):
        if z_i_star < result_value:
            result_value = z_i_star
            result_index = i
    return result_value, result_index

def step_one(z_n_star):
    lowest_value, lowest_index = select_entering_min(z_n_star)
    if lowest_index is None:
        print("All elements of z_N^* are non-negative; solution is optimal!")
        return None
    return lowest_index
```

[7]:
```
entering_index = step_one(z_N_s)
entering_var = nonbasic[entering_index]
print(f"Entering variable: x_{entering_var}")
```

```
Entering variable: x_0
```

2.1.1 Step 3: Compute primal step direction

[8]:
```
def e_j(j, n):  # compute the jth unit vector (dimension n)
    return np.array([[0]] * j + [[1]] + (n-j-1) * [[0]])
```

Compute the step update direction: $\Delta x_{\mathcal{B}} = B^{-1} N e_j$

[9]:
```
nej = N @ e_j(entering_index, len(nonbasic))
print_matrix(nej, prefix="Ne_j = ")
B_inv = np.linalg.inv(B)
print_matrix(B_inv, prefix="B^{-1} = ")
delta_x = B_inv @ nej
print_matrix(delta_x, prefix=r"\Delta x_{\mathcal{B}} = ")
```

$$Ne_j = \begin{bmatrix} 1.0 \\ 2.0 \\ 0.0 \end{bmatrix}$$

$$B^{-1} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\Delta x_{\mathcal{B}} = \begin{bmatrix} 1.0 \\ 2.0 \\ 0.0 \end{bmatrix}$$

2.1.2 Steps 4 and 5: Compute primal step length

We need to satisfy $x^*_{\mathcal{B}} \geq t\Delta x_{\mathcal{B}}$. Return `None, infinity` if unbounded, and t, i otherwise.

[10]:
```
def compute_primal_step_length(x_B_s, delta_x):
    t = np.inf
    leaving_index = None
    for i in range(len(delta_x)):
        if delta_x[i][0] <= 0: continue
        max_t = x_B_s[i][0] / delta_x[i][0]
        if max_t < t:
            leaving_index = i
            t = max_t
    if leaving_index is None:
        print("Primal unbounded!")
    return t, leaving_index
```

[11]:
```
t, leaving_index = compute_primal_step_length(x_B_s, delta_x)
print(f"Step length t = {t}")
leaving_var = basic[leaving_index]
print(f"Leaving variable: x_{leaving_var}")
```

```
Step length t = 1.0
Leaving variable: x_2
```

2.1.3 Step 6: Compute dual step direction

$\Delta z_{\mathcal{N}} = -(B^{-1}N)^T e_i$

[12]:
```
b_inv_n = B_inv @ N
print_matrix(-np.transpose(b_inv_n), prefix="-(B^{-1}N)^T = ")
delta_z = -np.transpose(b_inv_n) @ e_j(leaving_index, len(basic))
print_matrix(delta_z, prefix="\Delta z_{\mathcal{N}} = ")
```

$$-(B^{-1}N)^T = \begin{bmatrix} -1.0 & -2.0 & -0.0 \\ 1.0 & 1.0 & -1.0 \end{bmatrix}$$

$$\Delta z_{\mathcal{N}} = \begin{bmatrix} -1.0 \\ 1.0 \end{bmatrix}$$

2.1.4 Step 7: Dual step length

[13]:
```
def compute_dual_step_length(z_N_s, delta_z, entering_index):
    return z_N_s[entering_index][0] / delta_z[entering_index][0]
```

[14]:
```
s = compute_dual_step_length(z_N_s, delta_z, entering_index)
print(f"Dual step length s = {s}")
```

```
Dual step length s = 4.0
```

2.1.5 Step 8, 9: Update solutions/basis

[15]:
```
z_N_s -= s * delta_z
z_N_s[entering_index] = s
x_B_s -= t * delta_x
x_B_s[leaving_index] = t
basic[leaving_index] = entering_var
nonbasic[entering_index] = leaving_var
print_matrix(x_B_s, prefix="x_{\mathcal{B}}^* = ")
print_matrix(z_N_s, prefix="z_{\mathcal{N}}^* = ")
print("Basic: ", basic, ", non-basic: ", nonbasic)
print("Objective value: ", np.transpose(c[basic, :]) @ B_inv @ b)
```

$$x_{\mathcal{B}}^* = \begin{bmatrix} 1.0 \\ 1.0 \\ 5.0 \end{bmatrix}$$

$$z_{\mathcal{N}}^* = \begin{bmatrix} 4.0 \\ -7.0 \end{bmatrix}$$

```
Basic:  [0, 3, 4] , non-basic:  [2, 1]
Objective value:  [[4.]]
```

3 Iteration 2

3.0.1 N, B with new basis

[16]:
```
B = get_columns(A, basic)
N = get_columns(A, nonbasic)
B_inv = np.linalg.inv(B)
print_matrix(B, "B = ")
print_matrix(B_inv, "B^{-1} = ")
print_matrix(N, "N = ")
```

$$B = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 2.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$B^{-1} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -2.0 & 1.0 & -0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$N = \begin{bmatrix} 1.0 & -1.0 \\ 0.0 & -1.0 \\ 0.0 & 1.0 \end{bmatrix}$$

3.0.2 Step 1,2: Entering variable selection/optimality check

[17]:
```
entering_index = step_one(z_N_s)
entering_var = nonbasic[entering_index]
print(f"Entering variable: x_{entering_var}")
```

```
Entering variable: x_1
```

3.0.3 Step 3: Primal step direction

[18]:
```
nej = N @ e_j(entering_index, len(nonbasic))
print_matrix(nej, prefix="Ne_j = ")
delta_x = B_inv @ nej
print_matrix(delta_x, prefix=r"\Delta x_{\mathcal{B}} = ")
```

$$Ne_j = \begin{bmatrix} -1.0 \\ -1.0 \\ 1.0 \end{bmatrix}$$

$$\Delta x_{\mathcal{B}} = \begin{bmatrix} -1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

3.0.4 Step 4, 5: Primal step length, leaving variable

[19]:
```
t, leaving_index = compute_primal_step_length(x_B_s, delta_x)
print(f"Step length t = {t}")
leaving_var = basic[leaving_index]
print(f"Leaving variable: x_{leaving_var}")
```

```
Step length t = 1.0
Leaving variable: x_3
```

3.0.5 Step 6: Dual step direction

[20]:
```
b_inv_n = B_inv @ N
print_matrix(-np.transpose(b_inv_n), prefix="-(B^{-1}N)^T = ")
delta_z = -np.transpose(b_inv_n) @ e_j(leaving_index, len(basic))
print_matrix(delta_z, prefix="\Delta z_{\mathcal{N}} = ")
```

$$-(B^{-1}N)^T = \begin{bmatrix} -1.0 & 2.0 & -0.0 \\ 1.0 & -1.0 & -1.0 \end{bmatrix}$$

$$\Delta z_{\mathcal{N}} = \begin{bmatrix} 2.0 \\ -1.0 \end{bmatrix}$$

3.0.6 Step 7: Dual step length

[21]:
```
s = compute_dual_step_length(z_N_s, delta_z, entering_index)
print(f"Dual step length s = {s}")
```

```
Dual step length s = 7.0
```

3.0.7 Steps 8,9: Basis and solution update

[22]:
```
z_N_s -= s * delta_z
z_N_s[entering_index] = s
x_B_s -= t * delta_x
x_B_s[leaving_index] = t
basic[leaving_index] = entering_var
nonbasic[entering_index] = leaving_var
print_matrix(x_B_s, prefix="x_{\mathcal{B}}^* = ")
print_matrix(z_N_s, prefix="z_{\mathcal{N}}^* = ")
print("Basic: ", basic, ", non-basic: ", nonbasic)
print("Objective value: ", np.transpose(c[basic, :]) @ B_inv @ b)
```

$$x^*_{\mathcal{B}} = \begin{bmatrix} 2.0 \\ 1.0 \\ 4.0 \end{bmatrix}$$

$$z^*_{\mathcal{N}} = \begin{bmatrix} -10.0 \\ 7.0 \end{bmatrix}$$

```
Basic:  [0, 1, 4] , non-basic:  [2, 3]
Objective value:  [[7.]]
```

4 Iteration 3

4.0.1 N, B with new basis

[23]:
```
B = get_columns(A, basic)
N = get_columns(A, nonbasic)
B_inv = np.linalg.inv(B)
print_matrix(B, "B = ")
print_matrix(B_inv, "B^{-1} = ")
print_matrix(N, "N = ")
```

$$B = \begin{bmatrix} 1.0 & -1.0 & 0.0 \\ 2.0 & -1.0 & 0.0 \\ 0.0 & 1.0 & 1.0 \end{bmatrix}$$

$$B^{-1} = \begin{bmatrix} -1.0 & 1.0 & 0.0 \\ -2.0 & 1.0 & 0.0 \\ 2.0 & -1.0 & 1.0 \end{bmatrix}$$

$$N = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \\ 0.0 & 0.0 \end{bmatrix}$$

4.0.2 Step 1,2: Entering variable selection/optimality check

[24]:
```
entering_index = step_one(z_N_s)
entering_var = nonbasic[entering_index]
print(f"Entering variable: x_{entering_var}")
```

```
Entering variable: x_2
```

4.0.3 Step 3: Primal step direction

[25]:
```
nej = N @ e_j(entering_index, len(nonbasic))
print_matrix(nej, prefix="Ne_j = ")
delta_x = B_inv @ nej
print_matrix(delta_x, prefix=r"\Delta x_{\mathcal{B}} = ")
```

$$Ne_j = \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

$$\Delta x_{\mathcal{B}} = \begin{bmatrix} -1.0 \\ -2.0 \\ 2.0 \end{bmatrix}$$

4.0.4 Step 4, 5: Primal step length, leaving variable

[26]:
```
t, leaving_index = compute_primal_step_length(x_B_s, delta_x)
print(f"Step length t = {t}")
leaving_var = basic[leaving_index]
print(f"Leaving variable: x_{leaving_var}")
```

```
Step length t = 2.0
Leaving variable: x_4
```

4.0.5 Step 6: Dual step direction

[27]:
```
b_inv_n = B_inv @ N
print_matrix(-np.transpose(b_inv_n), prefix="-(B^{-1}N)^T = ")
delta_z = -np.transpose(b_inv_n) @ e_j(leaving_index, len(basic))
print_matrix(delta_z, prefix="\Delta z_{\mathcal{N}} = ")
```

$$-(B^{-1}N)^T = \begin{bmatrix} 1.0 & 2.0 & -2.0 \\ -1.0 & -1.0 & 1.0 \end{bmatrix}$$

$$\Delta z_{\mathcal{N}} = \begin{bmatrix} -2.0 \\ 1.0 \end{bmatrix}$$

4.0.6 Step 7: Dual step length

[28]:
```
s = compute_dual_step_length(z_N_s, delta_z, entering_index)
print(f"Dual step length s = {s}")
```

```
Dual step length s = 5.0
```

4.0.7 Steps 8,9: Basis and solution update

[29]:
```
z_N_s -= s * delta_z
z_N_s[entering_index] = s
x_B_s -= t * delta_x
x_B_s[leaving_index] = t
basic[leaving_index] = entering_var
nonbasic[entering_index] = leaving_var
print_matrix(x_B_s, prefix="x_{\mathcal{B}}^* = ")
print_matrix(z_N_s, prefix="z_{\mathcal{N}}^* = ")
print("Basic: ", basic, ", non-basic: ", nonbasic)
print("Objective value: ", np.transpose(c[basic, :]) @ B_inv @ b)
```

$$x_{\mathcal{B}}^* = \begin{bmatrix} 4.0 \\ 5.0 \\ 2.0 \end{bmatrix}$$

$$z_{\mathcal{N}}^* = \begin{bmatrix} 5.0 \\ 2.0 \end{bmatrix}$$

```
Basic:  [0, 1, 2] , non-basic:  [4, 3]
Objective value:  [[11.]]
```

5 Iteration 4

[30]:
```
B = get_columns(A, basic)
N = get_columns(A, nonbasic)
B_inv = np.linalg.inv(B)
print_matrix(B, "B = ")
print_matrix(B_inv, "B^{-1} = ")
print_matrix(N, "N = ")
```

$$B = \begin{bmatrix} 1.0 & -1.0 & 1.0 \\ 2.0 & -1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$$

$$B^{-1} = \begin{bmatrix} 0.0 & 0.5 & 0.5 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & -0.5 & 0.5 \end{bmatrix}$$

$$N = \begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 1.0 \\ 1.0 & 0.0 \end{bmatrix}$$

[31]:
```
entering_index = step_one(z_N_s)
print("Objective value: ", np.transpose(c[basic, :]) @ B_inv @ b)
```

```
All elements of z_N^* are non-negative; solution is optimal!
Objective value:  [[31.]]
```

[32]:
```
print_matrix(B_inv @ N)
```

$$\begin{bmatrix} 0.5 & 0.5 \\ 1.0 & 0.0 \\ 0.5 & -0.5 \end{bmatrix}$$

5.0.1 Optimality!

6 Whole procedure as method

[33]:
```
def primal_simplex(A, b, c):
    """
    Assumes that the matrix is given with the
    slack variables in the last m positions.
    """
    m, n = A.shape
    n -= m
    basic = list(range(n, n+m))
    nonbasic = list(range(n))
    iteration = 0
    x_B_s = np.array(b, dtype=np.float64)
    z_N_s = -np.array(c, dtype=np.float64)[:len(nonbasic)]
    while True:
        iteration += 1
        print("Iteration", iteration)
        # update B, N, B^-1, compute objective
        B = A[:,basic]
        N = A[:,nonbasic]
        B_inv = np.linalg.inv(B)
        oval = (np.transpose(c[basic, :]) @ B_inv @ b)[0][0]
        # compute entering index, check optimality
        entering_index = step_one(z_N_s)
        if entering_index is None:
            print("Optimality reached; optimal objective value: ", oval)
            primal = [0] * (n + m)
            for basic_index, basic_var in enumerate(basic):
                primal[basic_var] = x_B_s[basic_index]
            return oval, primal
        # compute primal step direction
        entering_var = nonbasic[entering_index]
        print(f"Not optimal yet; objective = {oval}, entering variable:␣
↪x_{entering_var}")
```

```
        nej = N @ e_j(entering_index, len(nonbasic))
        delta_x = B_inv @ nej
        # compute primal step length, leaving variable
        t, leaving_index = compute_primal_step_length(x_B_s, delta_x)
        if leaving_index is None:
            return np.inf, None
        leaving_var = basic[leaving_index]
        print(f"Leaving variable: x_{leaving_var}")
        # compute dual step direction
        delta_z = -np.transpose(B_inv @ N) @ e_j(leaving_index, len(basic))
        # compute dual step length
        s = compute_dual_step_length(z_N_s, delta_z, entering_index)
        # update primal and dual values, update basis
        z_N_s -= s * delta_z
        z_N_s[entering_index] = s
        x_B_s -= t * delta_x
        x_B_s[leaving_index] = t
        basic[leaving_index] = entering_var
        nonbasic[entering_index] = leaving_var
        print_matrix(x_B_s, prefix="x_{\mathcal{B}}^* = ")
        print_matrix(z_N_s, prefix="z_{\mathcal{N}}^* = ")
```

[34]:
```
objective, solution = primal_simplex(A, b, c)
```

```
Iteration 1
Not optimal yet; objective = 0.0, entering variable: x_0
Leaving variable: x_2
```

$$x_{\mathcal{B}}^* = \begin{bmatrix} 1.0 \\ 1.0 \\ 5.0 \end{bmatrix}$$

$$z_{\mathcal{N}}^* = \begin{bmatrix} 4.0 \\ -7.0 \end{bmatrix}$$

```
Iteration 2
Not optimal yet; objective = 4.0, entering variable: x_1
Leaving variable: x_3
```

$$x_{\mathcal{B}}^* = \begin{bmatrix} 2.0 \\ 1.0 \\ 4.0 \end{bmatrix}$$

$$z_{\mathcal{N}}^* = \begin{bmatrix} -10.0 \\ 7.0 \end{bmatrix}$$

```
Iteration 3
Not optimal yet; objective = 11.0, entering variable: x_2
Leaving variable: x_4
```

$$x_{\mathcal{B}}^* = \begin{bmatrix} 4.0 \\ 5.0 \\ 2.0 \end{bmatrix}$$

$$z_{\mathcal{N}}^* = \begin{bmatrix} 5.0 \\ 2.0 \end{bmatrix}$$

```
Iteration 4
All elements of z_N^* are non-negative; solution is optimal!
Optimality reached; optimal objective value:  31.0
```

[35]:
```
print_matrix(solution, prefix="x^* = ")
```

$$x^* = \begin{bmatrix} 4.0 \\ 5.0 \\ 2.0 \\ 0 \\ 0 \end{bmatrix}$$

[36]:
```
c2 = np.array([[3], [4], [-2], [0], [0]])
b2 = np.array([[2], [3]])
A2 = np.array([[1, 0.5, -5, 1, 0], [2, -1, 3, 0, 1]])
print_matrix(A2, prefix="A_2 = ")
print_matrix(b2, prefix="b_2 = ")
print_matrix(c2, prefix="c_2 = ")
```

$$A_2 = \begin{bmatrix} 1.0 & 0.5 & -5.0 & 1.0 & 0.0 \\ 2.0 & -1.0 & 3.0 & 0.0 & 1.0 \end{bmatrix}$$

$$b_2 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$c_2 = \begin{bmatrix} 3 \\ 4 \\ -2 \\ 0 \\ 0 \end{bmatrix}$$

[37]:
```
primal_simplex(A2, b2, c2)
```

```
Iteration 1
Not optimal yet; objective = 0.0, entering variable: x_1
Leaving variable: x_3
```

$$x_{\mathcal{B}}^* = \begin{bmatrix} 4.0 \\ 7.0 \end{bmatrix}$$

$$z_{\mathcal{N}}^* = \begin{bmatrix} 5.0 \\ 8.0 \\ -38.0 \end{bmatrix}$$

```
Iteration 2
Not optimal yet; objective = 16.0, entering variable: x_2
Primal unbounded!
```

[37]: (inf, None)