Report of Programming Task 2 of the course "Introduction to Optimization" - Fall 2024

Nikita Zagainov, Ilyas Galiev, Arthur Babkin, Nikita Menshikov, Sergey Aitov

September 2024

1 Team Information

- Team leader: Nikita Zagainov — 5

  Managed team work, Contributed to the algorithm implementation, Wrote report
- Team member 1: Ilyas Galiev — 5

  Contributed to the algorithm implementation
- Team member 2: Arthur Babkin — 5

  Contributed to the algorithm implementation, Wrote QA tests
- Team member 3: Nikita Menshikov — 5

  Contributed to the algorithm implementation, Adapted problem from previous assignment
- Team member 4: Sergey Aitov — 5

  Contributed to the algorithm implementation, Contributed to QA testing

2 Link to the product

Project source code

3 Programming language

Python

4 Linear programming problem

We aim to maximize nutritious value of salad given constraints on cost of its ingredients, maximum fats concentration, and weight of each individual component

Ingredient	Tomato	Cucumber	Bell Pepper	Lettuce Leaf	Onion
Cost, rub/kg	130	100	155	85	50
Nutritious value, ckal/kg	200	160	260	150	400
Max weight in salad, kg	0.6	0.6	0.6	0.2	0.05
Fats, proportion	0.004	0.005	0.006	0.003	0.004

Таблица 1: Ingredients and their properties

- Our problem is maximization problem
- Objective function & constraints:

$$\text{maximize } c^T x$$

subject to

$$Ax \leq b$$

where:

$$A = \begin{bmatrix} 130 & 100 & 155 & 85 & 50 \\ 0.004 & 0.005 & 0.006 & 0.003 & 0.004 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} 200 \\ 1 \\ 0.6 \\ 0.6 \\ 0.6 \\ 0.2 \\ 0.05 \end{bmatrix}$$

$$c = \begin{bmatrix} 200 \\ 160 \\ 260 \\ 150 \\ 400 \end{bmatrix}$$

However, since the algorithm we implemented solves maximization problem

$$\text{maximize } c^T x$$

subject to

$$Ax = b$$

We manually introduce slack variables to convert inequality constraints to equality constraints.

5 Output & Results

We tested our implementation of interior point method by comparing its outputs with scipy implementation, and all tests show that outputs of both methods are the same on multiple tests, including original problem.
The method is applicable to our problem:

$$\begin{aligned} &\text{Problem is bounded: True} \\ &x : [0.2115,\ 0.6,\ 0.6,\ 0.2,\ 0.05] \\ &f : 344.3 \end{aligned}$$

The results match with our previous simplex method implementation.

6 Code

```python
import numpy as np
from typing import Tuple


def pivot_col(tableau: np.ndarray, tol: float) -> int:
    last_row = tableau[-1, :-1]
    if np.all(last_row >= -tol):
        return -1
    return np.argmin(last_row)


def pivot_row(tableau: np.ndarray, tol: float, col: int) -> int:
    rhs = tableau[:-1, -1]
    lhs = tableau[:-1, col]
```

```
    ratios = np.full_like(rhs, np.inf)
    valid = lhs > tol
    ratios[valid] = rhs[valid] / lhs[valid]
    if np.all(ratios == np.inf):
        return -1
    return np.argmin(ratios)


def find_basic_solution(
    A: np.ndarray, b: np.ndarray, tol: float = 1e-6
) -> Tuple[bool, np.ndarray]:
    """
    Performs Phase I of the simplex method to find a basic feasible solution.

    Args:
        A: Coefficient matrix of the constraints (m x n).
        b: Right-hand side vector of the constraints (m,).
        tol: Tolerance for determining feasibility.

    Returns:
        A tuple containing:
        - A boolean indicating whether a feasible solution was found.
        - A numpy array representing the basic feasible solution (if found).
    """
    m, n = A.shape
    A_phase1 = np.hstack([A, np.eye(m)])
    c_phase1 = np.concatenate([np.zeros(n), np.ones(m)])
    B = list(range(n, n + m))

    tableau = np.hstack([A_phase1, b.reshape(-1, 1)])
    tableau = np.vstack([tableau, np.concatenate([c_phase1, [0]])])

    for i in range(m):
        tableau[-1, :] -= tableau[i, :]

    while True:
        col = np.argmin(tableau[-1, :-1])
        if tableau[-1, col] >= -tol:
            break

        ratios = []
        for i in range(m):
            if tableau[i, col] > tol:
                ratio = tableau[i, -1] / tableau[i, col]
                ratios.append((ratio, i))
        if not ratios:
            return False, None
        _, row = min(ratios)
```

```python

        pivot = tableau[row, col]
        tableau[row, :] /= pivot
        for i in range(m + 1):
            if i != row:
                tableau[i, :] -= tableau[i, col] * tableau[row, :]

        B[row] = col

    basic_solution = np.zeros(n + m)
    basic_solution[B] = tableau[:m, -1]
    if np.any(basic_solution[n:] > tol):
        return False, None

    x_basic = basic_solution[:n]
    x_basic = x_basic + 2e-5 # this is to avoid zero gradients on first step
    return True, x_basic


def interior_point(
    A: np.ndarray,
    b: np.ndarray,
    c: np.ndarray,
    alpha: float = 0.5,
    tol: float = 1e-6,
    max_iters: int = 100000,
) -> Tuple[bool, np.ndarray]:
    """
    Performs the interior point method to solve a linear programming problem.

    Args:
        A: Coefficient matrix of the constraints (m x n).
        b: Right-hand side vector of the constraints (m,).
        c: Coefficient vector of the objective function to be maximized (n,).
        alpha: Step size for the Newton step.
        tol: Tolerance for determining convergence.
        max_iters: Maximum number of iterations to perform.

    Returns:
        A tuple containing:
        - A boolean indicating whether a feasible solution was found.
        - A numpy array representing the optimal solution (if found).
    """
    bounded, x = find_basic_solution(A, b)
    if not bounded:
        return False, None
    prev_x = None
```

```python
    n_iters = 0

    while prev_x is None or np.linalg.norm(x - prev_x) > tol:
        n_iters += 1
        if n_iters > max_iters:
            return False, None, None

        D = x.copy()
        x_hat = x * (1 / D)
        A_hat = A * D
        c_hat = c * D
        P = np.eye(A_hat.shape[1]) - A_hat.T @ np.linalg.inv(A_hat @ A_hat.T) @ A_hat

        c_proj = P @ c_hat
        nu = np.abs(np.min(c_proj)) if np.any(c_proj < 0) else 1
        x_hat = x_hat + (alpha / nu) * c_proj
        prev_x = x.copy()
        x = D * x_hat

    return True, x, c @ x


def main():
    np.set_printoptions(precision=4, suppress=True)

    A = np.array(
        [
            [130, 100, 155, 85, 50],
            [0.004, 0.005, 0.006, 0.003, 0.004],
            [1, 0, 0, 0, 0],
            [0, 1, 0, 0, 0],
            [0, 0, 1, 0, 0],
            [0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1],
        ],
        dtype=np.float32,
    )
    b = np.array([200, 0.01, 0.6, 0.6, 0.6, 0.2, 0.05], dtype=np.float32)
    c = np.array([200, 160, 260, 150, 400], dtype=np.float32)

    # since we are maximizing c.T @ x s. t. A @ x <= b, we need to introduce slack variables:
    A_slack = np.hstack([A, np.eye(A.shape[0])])
    c_slack = np.concatenate([c, np.zeros(A.shape[0])])

    feasible, x, f = interior_point(A_slack, b, c_slack)
    if feasible:
        print(f"Optimal x: {x[: c.shape[0]]}, optimal value: {f}")
    else:
```

```
        print("No feasible solution found.")


if __name__ == "__main__":
    main()
```