Report 1

Team information.

- Team leader: Ruslan Belkov
- Team member 1: Eldar Mametov
- Team member 2: Mukhammadrizo Maribjonov
- Team member 3: Olga Puzhalina
- Team member 4: Kuklin Pavel
- Team member 5: Ivan Smirnov

Link to the product.

- The product is available: dantetemplar/simplex (github.com)

Programming language.

- Programming language: Python version 3.11+

Linear programming problem.

- Maximization or Minimization? Maximization
- Objective function:

$$z = 3x_1 + 4x_2 + 6x_3$$

- Constraint functions:

$$\begin{cases} -2x_1 + 2x_2 + 15x_3 \leq 12 \\ 11x_1 + 6x_2 + 14x_3 \leq 30 \\ 3x_1 - 8x_2 + 1x_3 \leq 24 \end{cases}$$

Input

The input contains:

- A vector of coefficients of objective function - C.
- A matrix of coefficients of constraint function - A.
- A vector of right-hand side numbers - b.
- The approximation accuracy ϵ.

Output/Results

The output contains:

- The string "The method is not applicable!"

  or

- A vector of decision variables - X^*.

- Maximum (minimum) value of the objective function.

Code

```
    from collections.abc import Collection
from typing import Optional

import numpy as np
import pandas as pd
import logging
from dataclasses import dataclass


logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)


ObjectiveCoefficients = Collection[float]
"""A vector of coefficients of the objective function"""
ConstraintCoefficients = Collection[Collection[float]]
"""A matrix of coefficients of the constraints"""
RightHandSides = Collection[float]
"""A vector of right-hand sides of the constraints"""


@dataclass
class Solution:
    """
    A solution of the linear programming problem.
    """

    f: float
    """The value of the objective function"""
    x: dict[str, float]
    """The values of the variables"""
    C: Optional[ObjectiveCoefficients] = None
    """The coefficients of the objective function"""
    A: Optional[ConstraintCoefficients] = None
    """The coefficients of the constraints"""
    b: Optional[RightHandSides] = None
    """The right-hand sides of the constraints"""

    def __str__(self):
        result_string = []

        if self.C is not None:
            result_string.append("Objective function:")
            objective = [f"{c}*x{i}" for i, c in enumerate(self.C)]
            objective = " + ".join(objective)
            result_string.append(objective)

        if self.A is not None:
```

```
            result_string.append("Constraints:")
            for i, row in enumerate(self.A):
                constraint = [f"{c}*x{i}" for i, c in enumerate(row)]
                constraint = " + ".join(constraint)
                result_string.append(f"{constraint} <= {self.b[i]}")

        result_string.append("Solution:")
        result_string.append(f"f = {self.f}")
        result_string.append(", ".join([f"{k} = {v}" for k, v in self.x.items()]))

        return "\n".join(result_string)


class Tableau:
    _tableau: pd.DataFrame
    """The tableau with the coefficients of the problem"""

    @property
    def m(self):
        return self._tableau.values

    @property
    def z(self) -> pd.Series:
        """
        The row of the objective function. Without the solution column.
        """
        return self._tableau.iloc[-1, :-1]

    @property
    def solution(self) -> pd.Series:
        """
        The column of the solution.
        """
        return self._tableau.solution

    @property
    def f(self) -> float:
        """
        The value of the objective function.
        """
        return self.solution.iloc[-1]

    def is_optimal(self) -> bool:
        """
        Checks if the tableau is optimal.

        :return: True if the tableau is optimal, False otherwise
        """
        return np.all(self.z <= 0)

    def __init__(self, tableau: np.ndarray, targets: int, slack: int):
        targets = [f"x{i}" for i in range(targets)]
        slack = [f"s{i}" for i in range(slack)]
        self._tableau = pd.DataFrame(
            tableau, columns=targets + slack + ["solution"], index=slack + ["z"]
        )

    @classmethod
    def base_case_to_tableau(
        cls, C: ObjectiveCoefficients, A: ConstraintCoefficients, b: RightHandSides
    ) -> "Tableau":
```

```
"""
Converts the [#input_data]_ base case of linear programming problem to the tableau

Tableau form::

    +------------------------+
    |   Coeff-s  |  Solution  |
    +========================+
    | A  A  1  0 | b          |
    | A  A  0  1 | b          |
    | C  C  0  0 | 0          |
    +------------------------+

As an example, the following problem:

>>> C = [1, 2]
>>> A = [[1, 1], [1, -1]]
>>> b = [2, 1]

will be converted to the following tableau (with an added slack variable for each

>>> Tableau.base_case_to_tableau(C, A, b).m
array([[ 1.,  1.,  1.,  0.,  2.],
       [ 1., -1.,  0.,  1.,  1.],
       [ 1.,  2.,  0.,  0.,  0.]])

.. [#input_data] Base case of linear programming problem is a problem in the follo

    * All constraints are inequalities of the form :math:‘a_1 x_1 + a_2 x_2 + ...
    * Maximization problem of objective function :math:‘c_1 x_1 + c_2 x_2 + ... +
    * All variables are non-negative.

"""

C = np.array(C)
A = np.array(A)
b = np.array(b)

cnt_of_equations, cnt_of_targets = A.shape
cnt_of_slack = cnt_of_equations

if len(C) != cnt_of_targets:
    raise ValueError(
        f"Number of coefficients of the objective function ({len(C)}) "
        f"does not match the number of variables ({cnt_of_targets})"
    )

if len(b) != cnt_of_equations:
    raise ValueError(
        f"Number of right-hand sides ({len(b)}) "
        f"does not match the number of equations ({cnt_of_equations})"
    )

logger.info(f"{cnt_of_equations=}")
logger.info(f"{cnt_of_targets=}")

tableau: np.ndarray = np.zeros(
    (cnt_of_equations + 1, cnt_of_targets + cnt_of_slack + 1)
)
tableau[-1, :cnt_of_targets] = C
tableau[:-1, :cnt_of_targets] = A
```

```
        tableau[:-1, cnt_of_targets:-1] = np.eye(cnt_of_equations)
        tableau[:-1, -1] = np.array(b)
        return Tableau(tableau, targets=cnt_of_targets, slack=cnt_of_slack)

    def find_pivot_column(self) -> int:
        """
        Finds the pivot column in the tableau.

        :return: The index of the pivot column
        """

        _temp = self.z.copy()
        _temp[_temp <= 0] = np.inf
        return np.argmin(_temp)

    def find_pivot_row(self, pivot_column: int) -> int:
        """
        Finds the pivot row in the tableau.

        :param pivot_column: The index of the pivot column
        :return: The index of the pivot row
        """
        divisors = self._tableau.iloc[:-1, pivot_column]
        restrictions = np.divide(
            self._tableau.iloc[:-1, -1],
            self._tableau.iloc[:-1, pivot_column],
            out=np.full(len(divisors), np.inf),
            where=divisors > 0,
        )
        return np.argmin(restrictions)

    def is_pivot_column_solvable(self, pivot_column: int) -> bool:
        """
        Checks if the pivot column is solvable.

        :param pivot_column: The index of the pivot column
        :return: True if the pivot column is solvable, False otherwise
        """
        return np.any(self._tableau.iloc[:-1, pivot_column] > 0)

    def swap_variable(self, pivot_row, pivot_column):
        """
        Swaps the variable in the pivot row and pivot column.

        :param pivot_row: Pivot row with loose variable
        :param pivot_column: Pivot column with tight variable
        """

        pivot_value = self._tableau.iloc[pivot_row, pivot_column]
        self._tableau.iloc[pivot_row, :] /= pivot_value
        # swap indices of the pivot row and pivot column
        pivot_row_str = self._tableau.index[pivot_row]
        pivot_column_str = self._tableau.columns[pivot_column]
        logger.info(
            f"Loose {pivot_row_str}(row {pivot_row}) and Tight {pivot_column_str}(col {piv
        )

        self._tableau.rename(
            index={pivot_row_str: pivot_column_str, pivot_column_str: pivot_row_str},
            columns={pivot_column_str: pivot_row_str, pivot_row_str: pivot_column_str},
            inplace=True,
```

```python
        )

    def __repr__(self):
        return self._tableau.__repr__()


def solve_using_simplex_method(
    C: ObjectiveCoefficients,
    A: ConstraintCoefficients,
    b: RightHandSides,
    max_iterations: int = 1000,
    ftol: float = 1e-8,
) -> Solution:
    """
    Solves the linear programming problem using the simplex method.

    :param ftol: Tolerance of the objective function
    :param C: A vector of coefficients of the objective function
    :param A: A matrix of coefficients of the constraints
    :param b: A vector of right-hand sides of the constraints
    :param max_iterations: Maximum number of iterations
    :return: solution of the linear programming problem and the value of the objective fun

    Example:
    >>> C = [1, 1, 0]
    >>> A = [[-1, 1, 1], [1, 0, 0], [0, 1, 0]]
    >>> b = [2, 4, 4]
    >>> solution = solve_using_simplex_method(C, A, b)
    >>> solution.x
    {'s0': 2.0, 'x0': 4.0, 'x1': 4.0, 'z': -8.0}
    >>> solution.f
    8.0

    Example:
    >>> C = [1.2, 1.7] # z = 1.2x1 + 1.7x2
    >>> A = [[1, 0], [0, 1], [1, 1]] # x1 <= 3000, x2 <= 4000, x1 + x2 <= 5000
    >>> b = [3000, 4000, 5000]
    >>> solution = solve_using_simplex_method(C, A, b)
    >>> solution.x
    {'s1': 1000.0, 'x0': 2000.0, 'x1': 4000.0, 'z': -8000.0}
    >>> solution.f
    8000.0
    """

    tableau = Tableau.base_case_to_tableau(C, A, b)
    logger.info(f"Initial tableau:\n{tableau}")

    solved_tableau, delta_f, iteration = _simplex(
        tableau, max_iterations=max_iterations, ftol=ftol
    )
    logger.info(f"Solved in {iteration} iterations and error {delta_f}")

    f = -solved_tableau.f

    return Solution(f=f, x=dict(solved_tableau.solution), C=C, A=A, b=b)


def _simplex(
    tableau: Tableau, max_iterations: int, ftol: float
) -> tuple[Tableau, float, int]:
    """
```

```
    Solves the linear programming problem using the simplex method.

    :param tableau: problem in the tableau form (already with artificial and slack variabl
    :param max_iterations: maximum number of iterations, after which the algorithm will ra
    :return: solved tableau
    """
    iteration = 0
    f: float
    prev_f: float
    delta_f: float
    f = prev_f = tableau.f

    while not tableau.is_optimal():
        iteration += 1
        logger.info(f"Iteration {iteration}")

        if iteration > max_iterations:
            raise RuntimeError("Maximum number of iterations exceeded")

        pivot_column = tableau.find_pivot_column()
        pivot_row = tableau.find_pivot_row(pivot_column)

        if not tableau.is_pivot_column_solvable(pivot_column):
            logger.info(
                f"Unboundedness in iteration {iteration}: Column {pivot_column} has no pos
            )
            raise RuntimeError(
                "The problem is not solvable because of the unboundedness.",
            )

        # swap around pivot
        tableau.swap_variable(pivot_row=pivot_row, pivot_column=pivot_column)
        pivot_row_values = tableau.m[pivot_row, :]
        # perform row operations to make pivot column 0 except for pivot row (pivot row is
        for eq_i in range(tableau.m.shape[0]):
            if eq_i != pivot_row:
                delta_row = pivot_row_values * tableau.m[eq_i, pivot_column]
                tableau.m[eq_i, :] -= delta_row

        logger.info(f"Tableau:\n{tableau}")
        f = tableau.f
        delta_f = f - prev_f
        if abs(delta_f) < ftol:
            logger.info("Optimal solution found by tolerance")
            break
        prev_f = f
    delta_f = f - prev_f

    return tableau, delta_f, iteration


def get_cnts_of_variables(tableau: np.ndarray) -> tuple[int, int]:
    """Returns the number of slack variables and target variables in the tableau."""

    cnt_of_equations, cnt_of_variables = tableau.shape
    cnt_of_slack = cnt_of_equations - 1
    cnt_of_target = cnt_of_variables - cnt_of_slack - 1
    return cnt_of_slack, cnt_of_target


# def get_solution(tableau: np.ndarray) -> np.ndarray:
```

```
#     """
#     Extracts the solution from the tableau.
#
#     :param tableau: The solved tableau
#     :return: The solution of the linear programming problem (values of the variables)
#     """
#     cnt_of_slack, cnt_of_target = get_cnts_of_variables(tableau)
#
#     x = np.zeros(cnt_of_target)
#
#     for i in range(cnt_of_target):
#         indices = np.where(tableau[:, i] == 1)[0]
#         solutions_for_variable = tableau[indices, -1]
#         if len(solutions_for_variable) == 1:
#             x[i] = solutions_for_variable[0]
#         elif len(solutions_for_variable) == 0:
#             x[i] = 0
#         else:
#             raise RuntimeError("The tableau is not optimal")
#     return x


if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)

    C = [3, 4, 6]
    A = [[-2, 2, 15], [11, 6, 14], [3, -8, 1]]
    b = [12, 30, 24]

    solution = solve_using_simplex_method(C, A, b, max_iterations=100)
    print(solution)
```