LINEAR PROGRAMMING USING PYTHON

MADE BY: LEAH GIBSON

FOR MATH 510

A SHORT GUIDE TEACHING YOU HOW TO USE LINPROG IN THE PYTHON PACKAGE SCIPY TO SOLVE LINEAR OPTIMIZATION PROBLEMS.

Table of Contents

About This File

This guide was created for a project in Math 510 at Colorado State University and is based on the guide found here.

SECTION 1

Preliminaries & Set Up

This section walks through the initial installations that should be done in order to use SciPy and PuLP and provides a brief explanation of what SciPy and PuLP are and the differences between them.

1.1 SciPy

SciPy is a Python package used for optimization, integration, interpolation, eigenvalue problems, algebraic equations, and differential equations. For linear programming, the optimization and root-finding library called linprog will be used. This library provides a very approachable framework to solve basic linear programming problems, but it is important to note that linprog can only solve *minimization* problems.

1.2 PuLP

For more complex linear programming problems that require more fine tuning, there is a different Python package called PuLP that can be used. PuLP can solve both minimization and maximization problems and can also tackle inter and mixed integer linear programs as is demonstrated in this document.

1.3 Installation

SciPy is easy to install using pip. Open the terminal and enter the following command:

```
>>> python -m pip install scipy
```

or

```
>>> python3 -m pip install scipy
```

depending on the version of Python being used. (I am running Python 3.10.7 and thus used the later of the lines.)

We take a similar approach to installing PuLP:

```
>>> python3 - m pip install pulp
```

If you are running Linux or Mac, you may also want to run

```
>>> pulptest
```

which enables the default solvers for PuLP. Now we are ready to begin coding!

SECTION 2

SciPy

This section walks through how to solve a simple linear optimization problem in SciPy by way of an example.

2.1 The Mathematical Problem

We begin with the following problem (which may look familiar to those of you in Math 510 this semester!):

$$\begin{aligned} \text{Maximize } z = x + y \text{ for } x, y \in \mathbb{R} \\ \text{satisfying } -x + y \leq 1 \\ x + 6y \leq 15 \\ 4x - y \leq 10 \\ x \geq 0 \\ y \geq 0 \end{aligned}$$

Visually, we are searching for a solution in the white region shown in Figure 1.

Recall `linprog` can only solve minimization problems, specifically of the form

$$\begin{aligned} \min_x c^T x \\ \text{such that } A_{ub} x \leq b_{ub} \\ A_{eq} x = b_{eq} \\ l \leq x \leq u \end{aligned}$$

where A_{ub} and A_{eq} are matricies, b_{ub} and b_{eq} are vectors, and l and u and the lower and upper bounds of each variable. We must first turn this maximization problem into a minimization

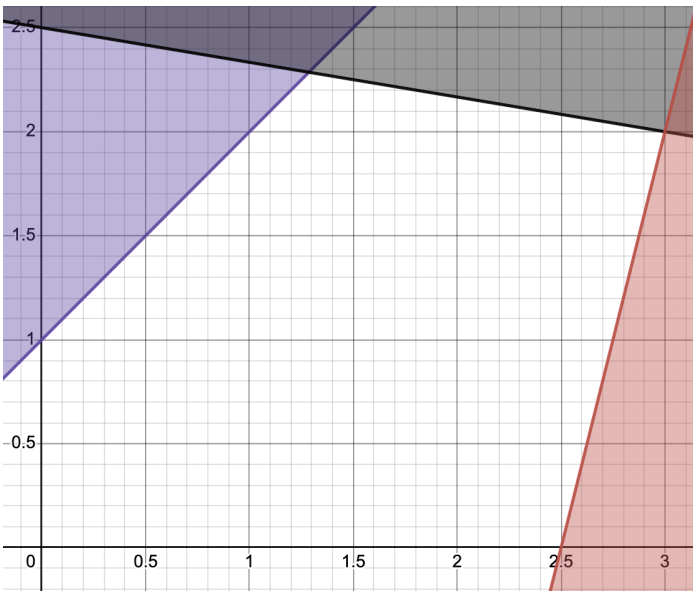

Figure 1: The feasible solution for the maximization problem.

problem and then turn this math into something that `linprog` can understand. It is relatively easy to turn a maximization problem into a minimization problem:

$$\begin{aligned} \text{Maximize } -z = -x - y \text{ for } x, y \in \mathbb{R} \\ \text{satisfying } -x + y \leq 1 \\ x + 6y \leq 15 \\ 4x - y \leq 10 \\ x \geq 0 \\ y \geq 0 \end{aligned}$$

2.2 Coding the Problem

To solve optimization problems using SciPy, we first import `linprog` from SciPy's optimization and root finding library using the line

```
>>> from scipy.optimize import linprog
```

2.2.1 The Objective Function and Constraints

Now we can set up the problem. We first define the input variables as vectors and matrices where each entry corresponds to a coefficient from the optimization problem. We begin with the objective function $-z = -x + -y$ which translates to

```
>>> obj = [-1, -1]
```

Next, we make a matrix whose coefficients correspond to the left hand side of the inequalities:

```
>>> lhs_inequality = [[-1, 1],
                      [1, 6],
                      [4, -1]]
```

Corresponding to this matrix is a vector consisting of the values on the right hand side of the inequalities. Special care must be taken to ensure the n^{th} row of the matrix corresponds to the n^{th} entry of the right hand side vector. For our example, this looks like:

```
>>> rhs_inequality = [1,
                      15,
                      10]
```

If there were any constraints consisting of equalities, this would be handled in the same manner as the inequalities, but could have the names "`lhs_equality`" and "`rhs_equality`" for example.

Lastly, we define the bounds for each variable. By default, linprog assumes the bounds of each variable are $[0,\infty)$, but we will explicitly write it out here to learn the proper syntax:

```
>>> bound = [(0, float("inf")),
             (0, float("inf"))]
```

Again, ensure the ordering is consistent with the ordering of the previous vectors and matrices. That is, if x is always the first entry, that should be the case here as well. Thus the first line corresponds to the bounds of x and the second line corresponds to the bounds of y.

2.2.2 Doing the Optimization

We are finally ready to do the optimization. We write this in the following way:

```
>>> optimization = linprog(c=obj,
                           A=lhs_inequality,
                           b_ub=rhs_inequality,
                           bounds=bound,
                           method=" ")
```

There are multiple choices for `method` and this will be discussed in a moment. In general, `linprog` would also accept the constraints from equality constraints in the following way:

```
>>> optimization = linprog(c=obj,
                           A_ub=lhs_inequality,
                           b_ub=rhs_inequality,
                           A_eq=lhs_equality,
                           b_eq=rhs_equality,
```

```
                bounds=bound,
                method=" ")
```

There are a few more options that linprog has to further fine tune the solving method. For more detail, see the documentation for linprog.

The parameter `method` is used to define the linear programming method used to solve the optimization problem. There are a number of options, but they slightly differ depending on the version of SciPy used. For SciPy 1.11.0 and newer, the methods are:

1. `method="highs"`
2. `method="highs-ds`
3. `method="highs-ipm`

The method `highs` is selected by default and will then choose between `highs-ds` (a simplex method) or `highs-ipm` (an interior point method) depending on the problem. In older versions of SciPy, the options are

1. `method="simplex"`
2. `method="revised simplex"`
3. `method="simplex"`

2.2.3 Optimization Output

To display the results of the optimization, include the line

```
>>> print(optimization)
```

The following explains the output of optimization if `method="highs"` is used. We will not cover the meaning of ever line, but each line is explained in the documentation.

A few of the lines report back if the solution was found successfully. The line

```
>>> success: True
```

is a Boolean that outputs `True` when the algorithm succeeds in finding an optimal solution. The line

```
>>> status: 0
```

provides a bit more information about the success (or not if `success: False`) of the program. As a whole, `status` returns an integer between 0 and 4, and each number provides different information. A status of 0 means the optimization terminated successfully. There is also a line

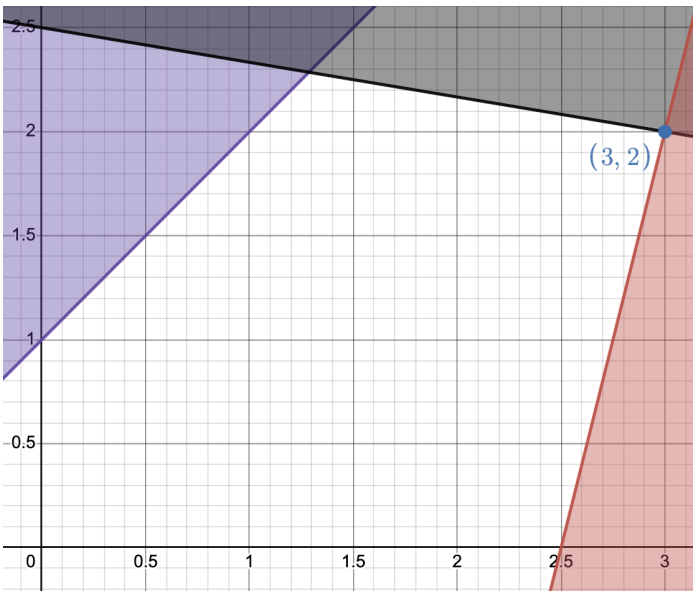

Figure 2: The optimal solution displayed in the feasible region.

```
>>> message: 'Optimization terminated successfully. (HiGHS Status 7: Optimal)'
```

which also provides a descriptor or the exit status of the algorithm.

Once it's been determined that the program successfully found a solution, the following two lines provide information about the solution. The line

```
>>> fun: -5.0
```

returns the value of the minimized function. So for us, $-x - y = -z = -5$ and the maximized value for the original problem is 5. The next line of importance is

```
>>> x: array([3., 2.])
```

which returns the value of each variable that minimizes the function. For our example, this means the, the solution is (x,y) = (3,2). These 5 lines are the primary lines to focus on when doing solving a basic linear optimization problem. The additional returns not explained here go into more depth about the algorithm's solving process. We'll next explore a scenario in which linear programming can be used to solve a real world problem.

SECTION 3

Example Application

Linear programming can be used in a variety of applications, some of which are in environmental quality control. The paper *Mathematical Programming Models for Environmental Quality Control* by Harvey J. Greenberg outlines the history and use of linear programming in environmental quality control.

Below is a made up problem showing one use case of linear programming in air quality.

3.1 The Problem

Suppose a factory wants to minimize the damage from air pollutants they release each day. This factory released carbon dioxide (CO_2), methane (CH_4), nitrogen oxides (NO_x's), and ammonia (NH_4) into the air. In order to determine the ideal emission amounts of each pollutant, the factory quantifies the damage from each pollutant using a penalty term proportional to the emissions of each gas in parts per million (ppm). The penalty terms are

- CO_2: 2.5 / 1 ppm
- CH_4: 5 / 1 ppm
- NO_x's: 7 / 1 ppm
- NH_3: 1 / 1 ppm

Mathematically, the factory's goal of minimizing the damage form their pollutants is given by the equation

$$\min_x 2.5x_1 + 5x_2 + 7x_3 + x_4$$

where x_1 is CO_2, x_2 is CH_4, x_3 is NO_x's, and x_4 is NH_3.

The company does have some constraints. They cannot cut back on their total production, and have determined the following combinations of pollutants must be used to keep up with production demands:

$$\begin{aligned} 3x_1 + 1x_4 &\geq 1 \\ x_2 + x_3 + 3x_4 &\geq 1.4 \\ x_2 + x_3 &\geq 0.3 \\ x_1 + 0.5x_3 &= 0.7 \end{aligned}$$

They factory is also under pressure from the EPA to meet the following maximum emission standards: they cannot emit more than 1 ppm of NO_x's on any given day and their total emissions cannot exceed 1.5 ppm. Mathematically,

$$\begin{aligned} x_3 &\leq 1 \\ x_1 + x_2 + x_3 + x_4 &\leq 1.5 \end{aligned}$$

3.2 The Set-up and Solution

This is a problem that can be solved using the method previously explained. As a linear optimization problem, the goal is to minimize

$$2.5x_1 + 5x_2 + 7x_3 + x_4$$

subject to the constraints

$$\begin{aligned} -3x_1 - x_4 &\leq -1 \\ -x_2 - x_3 - 3x_4 &\leq 1.4 \\ -x_2 - x_3 &\leq 0.3 \\ x_1 + x_2 + x_3 + x_4 &\leq 1.5 \\ x_3 &\leq 1 \\ x_1, x_2, x_3, x_4 &\geq 0 \end{aligned}$$

Just as before, the left hand side of these equations form matrices and the right hand side forms vector. The linear program terminated successfully and reveals that the ideal emissions for the least environmental impact are:

- CO_2: 0.7 ppm
- CH_4: 0.3 ppm
- NO_x's: 0 ppm

- NH_3: 0.37 ppm

Keep in mind that this was a made up problem, but given the knowledge of actual EPA regulations, the damage of different pollutants, and the needs to a factory and company, a linear program could be used to solve problems similar to this.
Next, we will explore linear programming using PuLP!

4

SECTION

PuLP

This section explores linear programming using PuLP. We will begin with the same problem that we had when using SciPy:

$$\begin{aligned} \text{Maximize } z = x + y \text{ for } x, y \in \mathbb{R} \\ \text{satisfying } -x + y \leq 1 \\ x + 6y \leq 15 \\ 4x - y \leq 10 \\ x \geq 0 \\ y \geq 0 \end{aligned}$$

One of the benefits of PuLP is that we will not need to change the problem to a minimization problem to solve.

4.1 Coding the Problem

To begin, we must import the following classes from PuLP:

```
>>> from pulp import LpMaximize, LpProblem, LpStatus, lpSum, LpVariable
```

We will not explore each of these currently, but as we work though the problem, the role of each should become apparent.
We first use `LpProblem` to make the model. Here, we specify the whether we are working with minimization problem (the default) or a maximization problem like we have here. We write the

line

```
>>> model = LpProblem(name="m510example", sense=LpMaximize)
```

You may pick the name of your problem. Use the `sense` parameter to specify the if the problem is a minimization problem (`LpMinimize`) or a maximization problem (`LpMaximize`).
Next, you can define the unknown variables and their domain in the problem using `LpVariable`

```
>>> x = LpVariable(name="x", lowBound = 0)
>>> y = LpVariable(name="y", lowBound = 0)
```

By default, the lower bound (`lowBound`) of each variable is set to negative infinity and the upper bound (`upBound`) is set to positive infinity. The next step is to imput the constraints and the objective function. In PuLP, there is no need to use matrices or vectors. Since we named the variables, PuLP allows us to use these names when imputing the constraints:

```
>>> model += (-1 * x + y <= 1, "first constraint")
>>> model += (x + 6 * y <= 15, "second constraint")
>>> model += (4 * x - y <= 10, "third constraint")
```

Each constraints is added to the model we named using `LpProblem`. It is optional to name the constraints. After entering all constraints, we lastly input the objective function:

```
>>> model += x + y
```

At this point, you can now run

```
>>> print(model)
```

to ensure the problem is set up correctly. Specifically for this problem, you will see the following output:

```
>>> m510example:
>>> MAXIMIZE
>>> 1*x + 1*y + 0
>>> SUBJECT TO
>>> first_constraint: - x + y <= 1

>>> second_constraint: x + 6 y <= 15

>>> third_constraint: 4 x - y <= 10

>>> VARIABLES
```

```
>>> x Continuous
>>> y Continuous
```

Now the only thing left to do is tell the machine to solve the problem. To do this, include the line

```
>>> status = model.solve()
```

To determine if the solver succeeded in finding a solution, use the line

```
>>> print(f"status: {model.status}, {LpStatus[model.status]}")
```

To see the output of the optimization function using the optimal variables, include

```
>>> print(f"objective: {model.objective.value()}")
```

To see the values of the variables, use the line

```
>>> for var in model.variables(): print(f"{var.name}: {var.value()}")
```

4.2 Integer and Mixed Integer Linear Programming

Suppose that instead of searching for any solution to a linear program, we want to specify that some or all of the unknown variables should be integers. Practically, this may be necessary if the things represented by these unknown variables are objects cannot exist as anything other than whole numbers. For example, maybe we need to know how to optimally organize people into groups. It is not possible to put half of a person into a group. In these scenarios, it is necessary to search for an optimal solution that is in integer. It is relatively easy to specify this when using PuLP. We will explore this using a new example since the previous example has an optimal solution that is an integer without specifying that this must be the case.

We will begin with integer linear programming. The new problem is thus

$$\begin{aligned} \text{Maximize } x+y \text{ for } x,y \in \mathbb{Z} & \\ \text{subject to } x-y &\leq 0 \\ \frac{5}{2}x-\frac{1}{3}y &\leq 7 \\ -x+2y &\leq 8 \\ x &\geq 0 \\ y &\geq 0 \end{aligned}$$

Visually, we are searching for a solution in the white region of Figure 3. If they were real numbers, both x and y should be as large as possible to maximize $x+y$ and we should expect the

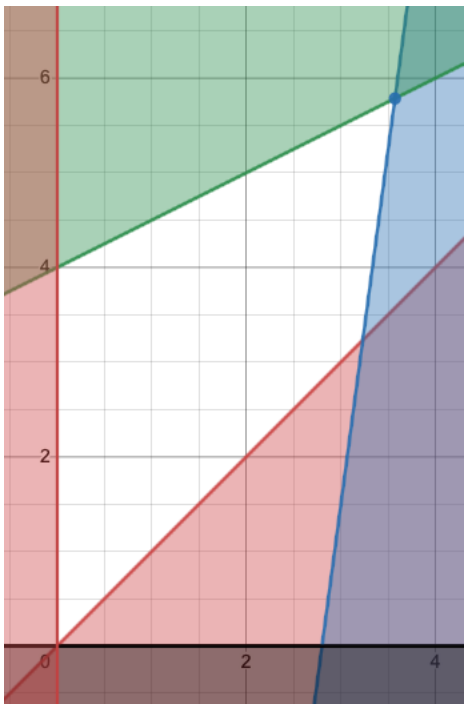

Figure 3: The feasible region and optimal solution if $x, y \in \mathbb{R}$.

solution to occur at a vertex. This is indeed the case, and the optimal variables (if the were in $\mathbb{R}$) are plotted as a blue dot. Let's see how this changed if they must be integers.

In PuLP, we specify that we'd like integer solutions in `LpVariable` by including `cat="Integer"`. Everything else is the same process as before, so we should code the following

```
# create the model
model = LpProblem(name="integer linear program", sense=LpMaximize)

# variables
x = LpVariable(name="x", lowBound = 0, cat = "Integer")
y = LpVariable(name="y", lowBound = 0, cat="Integer")

# constraints
model += (x - y <= 0, "first constraint")
model += ((5/2 )* x - (1/3) * y <= 7, "second constraint")
model += (-1 * x + 2 * y <= 8, "third constraint")

# objective function
objective_function = x + y

# add objective function to model
model += objective_function

print(model)
```

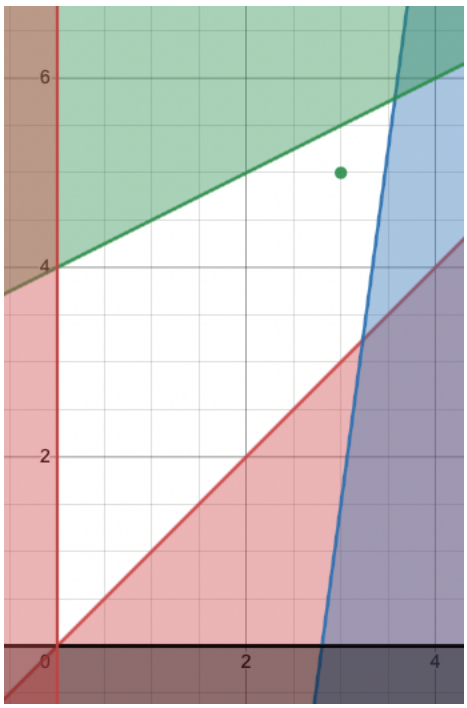

Figure 4: The feasible region and optimal solution if $x, y \in \mathbb{Z}$.

```
# solve the problem
status = model.solve()

# see the solution
print(f"status: {model.status}, {LpStatus[model.status]}")  # success?

print(f"objective: {model.objective.value()}") # output using optimal variables

for var in model.variables(): print(f"{var.name}: {var.value()}")  # print variables
```

The optimal values are now $x = 3$ and $y = 5$. Visually, this new solution is located inside the feasible region, but is near the optimal values when $x, y \in \mathbb{R}$. See Figure 4.

What if we were to do mixed integer linear programming and required that only x be an integer? Where could we expect this solution to lie? We would change the line for y to

```
y = LpVariable(name="y", lowBound = 0)
```

and keep everything else the same. This yields $x = 3$ and $y = 5.5$. This point lies on the line $-x + 2y = 8$. See Figure 5. If we were to switch these now and force y to be an integer, we would get that $x = 3.467$, $y = 5$, which lies on the line $\frac{5}{2}x - \frac{1}{3}y = 7$. See Figure 6.
Given that PuLP can handle both maximization and minimization problems and can easily be adjusted to do different types of linear programming, it can often be a better tool for solving linear programs.

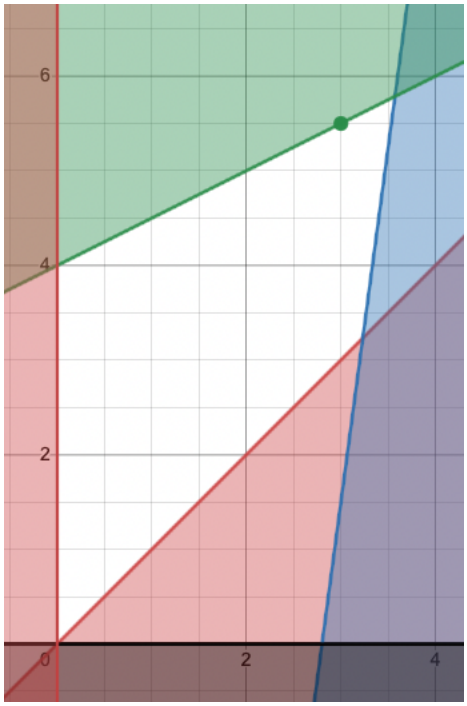

Figure 5: The feasible region and optimal solution if $x \in \mathbb{Z}$ and $y \in \mathbb{R}$.

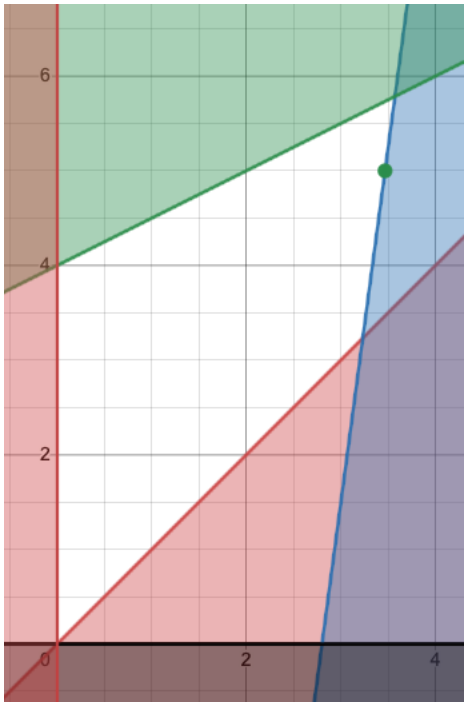

Figure 6: The feasible region and optimal solution if $x, y \in \mathbb{R}$.