CS 3430: SciComp with Py
Assignment 03
2D Linear Programming

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 27, 2024

Learning Objectives

1. 2D Linear Programming (LP)
2. Plotting 2D Linear Constraints
3. Determining Corner Points
4. Standard Maximum Problems
5. 2D Plots with `matplotlib`

Introduction

In this assignment, we'll implement a *semi*-automatic method of solving Linear Programming (LP) problems in 2D. The method is semi-automatic in the sense that we will determine the corner points of feasible sets by looking at the constraint line plots. Full automation will have to wait until next week when we learn the simplex algorithm to solve LP problems for any number of dimensions. In the meantime, this assignment will help us develop better geometric intuitions of how LP works.

You should not think that semi-automatic methods are restrictive or limiting. Data visualization through plotting is an integral part of scientific computing (and many other branches of science and engineering). When you as a practitioner or a researcher do not know what to do, the first thing to try is to plot your data and see if you can detect any patterns. A frequent recommendation I give to my students is – when in doubt, plot.

You will save your coding solutions in `cs3430_s24_hw03.py` included in the zip and submit it in Canvas.

Optional Problem 0: (0 points)

Review the PDF of Lectures 05 in Canvas/Announcements and/or your lecture notes. Make sure that you are comfortable with such concepts as *optimization problem*, *decision variable*, *objective function*, *constraint*, *feasible set*, *boundary line*, *half plane*, *bounded/unbounded set*, and *corner/extreme point*. We will go over these concepts one more time in the upcoming Lecture 06 on 01/29/2024. I also encourage you to take a look at and run the Python code in `graphs2D.py`, which I attached to the Lecture 05 PDF in Canvas, where I coded up several examples of how to plot functions with `matplotlib`.

If you find typos or inconsistencies in my slides/code samples/unit tests, please let me know. I am human, therefore I err. I always appreciate constructive feedback from my students. That's what learning is all about. A frequent pitfall for a teacher or a researcher (or, as is the case with me, a teaching researcher) is getting too close to the material to the point when it becomes your second nature and you assume that everybody is on the same page with you. So if you let me know how I can improve my slides/code samples/unit tests to make them more understandable, I will make an addendum (as I did regarding the computation of determinants without row scaling) and announce it in Canvas. This will help me improve my presentation methods for the current and future students of my this class.

Problem 1: (1 point)

Let us start with lines, which is the backbone of 2D LP. Any line in 2D can be represented as $Ax + By = C$, where A, B, and C are reals. Thus, we can represent lines as 3-tuples. For example, $4x + 3y = 480$ can be represented in Python as follows.

```
>>> line1 = (4.0, 3.0, 480.0)
```

We can then unpack the coefficients back into the variables A, B, C.

```
>>> A, B, C = line1
>>> A
4.0
>>> B
3.0
>>> C
480.0
```

We start by implementing the function `line_ip(line1, line2)` that takes 2 lines represented as 3-tuples and returns a 2×1 numpy vector `v` (i.e., a numpy array) that represents their intersection point (ip) so that `v[0,0]` is the x-coordinate of the intersection and `v[1,0]` is its y-coordinate.

If there is no intersection (i.e., the lines are parallel), the function returns `None`. In implementing this function, you can use `np.linalg.solve()` to solve $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, where the 2×2 matrix $\mathbf{A}$ consists of the coefficients of the two lines and $\mathbf{b}$ is a column vector of the two $\mathbf{C}$ coefficients. In other words, to find an intesection between 2 lines $a_1x + b_1y = c_1$ and $a_2x + b_2y = c_2$ is the same as solving the following linear system.

$$\begin{bmatrix} a_1 & b_1 & | & c_1 \\ a_2 & b_2 & | & c_2 \end{bmatrix}$$

Let us make it more concrete. If we want to find an intersection between $4x+3y = 480$ and $3x+6y = 720$, we can use Gauss-Jordan Elimination (GJE) that we covered in the first 2 lectures in order to solve the following linear system.

$$\begin{bmatrix} 4 & 3 & | & 480 \\ 3 & 6 & | & 720 \end{bmatrix}$$

So, let's do it with GJE.

```
>>> import numpy as np
>>> import numpy.linalg
>>> A = np.array([[4, 3],
                  [3, 6]],
                 dtype=float)
>>> b = np.array([[480],
                  [720]],
                 dtype=float)
>>> x = np.linalg.solve(A, b)
>>> x
array([[48.],
       [96.]])
>>> np.allclose(np.dot(A,x), b)
True
```

The above solution tells us that the two lines intersect at $(48, 96)$.

You can use the same approach in impelementing `line_ip()`. Below are a couple of test runs of my implementation `line_ip()`.

```
>>> from cs3430_s24_hw03 import *
>>> line1 = (4.0, 3.0, 480.0)
>>> line2 = (3.0, 6.0, 720.0)
>>> ip12 = line_ip(line1, line2)
>>> ip12
array([[48.],
       [96.]])
>>> ip21 = line_ip(line2, line1)
>>> ip21
array([[48.],
       [96.]])
>>> A = np.array([[4.0, 3.0], [3.0, 6.0]])
>>> b = np.array([[480.0], [720.0]])
>>> np.allclose(np.dot(A, ip12), np.dot(A, ip21))
True
>>> np.allclose(np.dot(A, ip12), b)
True
>>> np.allclose(np.dot(A, ip21), b)
True
>>> line3 = (4.0, 3.0, 200.0)
>>> line4 = (4.0, 3.0, 250.0)
>>> ip34 = line_ip(line3, line4)
>>> ip34 is None
True
```

Let us implement a function to check if the intersection point (if it is not `None`, of course) is correct at a given error level.

```
def check_line_ip(line1, line2, ip, err=0.0001):
    assert ip is not None
    A1, B1, C1 = line1
    A2, B2, C2 = line2
    x, y = ip[0, 0], ip[1, 0]
    assert np.allclose(abs((A1*x + B1*y) - C1), err)
    assert np.allclose(abs((A2*x + B2*y) - C2), err)
    return True
```

We can use `check_line_ip()` to verify what we've just verified with `np.dot()`.

```
>>> import numpy as np
>>> line1 = (4.0, 3.0, 480.0)
>>> line2 = (3.0, 6.0, 720.0)
>>> from cs3430_s24_hw03 import line_ip, check_line_ip
>>> check_line_ip(line1, line2, line_ip(line1, line2), err=0.0e-11)
True
```

Let us proceed to implement the function `find_line_ips(lines)` that takes an array of lines and returns a list of pairwise intersection points computed by `line_ip()`. For example, if `line1`, `line2`, and `line3` are 3-tuples representing lines, `find_line_ips([line1, line2, line3])` returns the intersection points between `line1` and `line2`, `line1` and `line3`, and `line2` and `line3`. Be careful not to compute the same intersection twice (e.g., between `line1` and `line2` and between `line2` and `line1`). Of course, computing duplicate intersections will not render the required computation incorrect, but it will make it less efficient. Here's a test run of my implementation.

```
>>> from cs3430_s24_hw03 import find_line_ips, check_line_ip
>>> line1 = (1.0, 0.0, 1.0)
>>> line2 = (1.0, -2.0, 0.0)
>>> line3 = (3.0, 4.0, 12.0)
>>> ips = find_line_ips([line1, line2, line3])
>>> ips
```

```
[array([[1. ],
        [0.5]]),
array([[1.  ],
       [2.25]]),
array([[2.4],
       [1.2]])]
>>> check_line_ip(line1, line2, ips[0], err=0.1e-11)
True
>>> check_line_ip(line2, line1, ips[0], err=0.1e-11)
True
>>> check_line_ip(line1, line3, ips[1], err=0.1e-11)
True
>>> check_line_ip(line3, line1, ips[1], err=0.1e-11)
True
>>> check_line_ip(line2, line3, ips[2], err=0.1e-11)
True
>>> check_line_ip(line3, line2, ips[2], err=0.1e-11)
True
```

As we discussed in Lectures 05 (and will continue to discuss in Lecture 06), we need to maximize functions to solve maximization problems. So, let us implement the function `max_obj_fun(f, points)` that takes an objective function `f` and maximizes it on a list of points, each of which is a 2×1 numpy array. To maximize a function on a list of points is to find a point where the function achieves a maximum value. All things being equal, ties are broken arbitrarily. The function returns a 2-tuple that consists of a point and the value of f at that point. Here's an example.

```
>>> from cs3430_s24_hw03 import find_line_ips, max_obj_fun
>>> line1 = (1.0, 0.0, 1.0)
>>> line2 = (1.0, -2.0, 0.0)
>>> line3 = (3.0, 4.0, 12.0)
>>> obj_fun = lambda x, y: 10.0*x + 5.0*y
>>> ips = find_line_ips([line1, line2, line3])
>>> m = max_obj_fun(obj_fun, ips)
>>> m
(array([[2.4],
       [1.2]]), 30.0)
```

The function `max_obj_fun()` returned a 2-tuple `(array([[2.4], [1.2]]), 30.0)`, the mathematical interpreation of which is that at the point $(2.4, 1.2)$ the objective function $f(x, y) = 10x + 5y$ achieves a maximum value of 30.0 on the list of the intersection points `ips` returned by `find_line_ips()`. We can verify it by applying `obj_fun` to each point in `ips`.

```
>>> obj_fun(ips[0][0,0], ips[0][1,0])
12.5
>>> obj_fun(ips[1][0,0], ips[1][1,0])
21.25
>>> obj_fun(ips[2][0,0], ips[2][1,0])
30.0
```

Now implement the function `min_obj_fun(f, points)` that takes an objective function `f` and minimizes it on a list of points, each of which is a 2×1 numpy array. To minimize a function on a list of points is to find a point where the function achieves a maximum value. All things being equal, ties should also be broken arbitrarily. The function returns a 2-tuple that consists of a point and the value of f at that point. Here's an example.

```
>>> from cs3430_s24_hw03 import find_line_ips, min_obj_fun
>>> line1 = (1.0, 0.0, 1.0)
>>> line2 = (1.0, -2.0, 0.0)
>>> line3 = (3.0, 4.0, 12.0)
>>> obj_fun = lambda x, y: 10.0*x + 5.0*y
```

```
>>> ips = find_line_ips([line1, line2, line3])
>>> m = min_obj_fun(obj_fun, ips)
>>> m
>>> from cs3430_s24_hw03 import find_line_ips, min_obj_fun
>>> line1 = (1.0, 0.0, 1.0)
>>> line2 = (1.0, -2.0, 0.0)
>>> line3 = (3.0, 4.0, 12.0)
>>> obj_fun = lambda x, y: 10.0*x + 5.0*y
>>> ips = find_line_ips([line1, line2, line3])
>>> ips
[array([[1. ], [0.5]]), array([[1.  ], [2.25]]),
 array([[2.4], [1.2]])]
>>> m = min_obj_fun(obj_fun, ips)
>>> m
(array([[1. ], [0.5]]), 12.5)
```

Coding Lab: (0 points)

After Problem 1, we have all everything in place for solving standard maximum problems (SMP's) in 2D semi-automatically. This coding lab will help us solve Problems 2, 3, and 4 below. Given a 2D SMP, the first step is to identify the objective function and the constraints. Once the constraints are identified, we can plot them to determine the corner points and compute their coordinates with `find_line_ips()`. Remember that not all intersection points between constraint lines are corner points of the feasible set. We really have to look at the plots and determine the feasible set and its corner points.

Let us consider again the Ted's Toys problem we analyzed in Lectures 05 (see the Lecture 05 PDF in Canvas and/or your class notes). Recall that this problem has the following constraints, where x is the number of toy cars and y is the number of toy trucks Ted's company makes per day.

1. $x \geq 0$;
2. $y \geq 0$;
3. plastic constraint: $4x + 3y \leq 480$;
4. steel constraint: $3x + 6y \leq 720$.

First, we have to define the plastic and steel constraints, which we will do by abstracting them as Python functions.

```
def plastic_constraint(x): return -(4/3.0)*x + 160.0
def steel_constraint(x): return -0.5*x + 120.0
```

Second, we need to generate the x and y values for both constraints to plot with `matplotlib`. Note that the lines $x = 0$ and $y = 0$ can be simply defined as the pair of points `[x1, x2]`, `[y1, y2]`. The upper bound of the linear space (i.e., 160) is problem-dependent. In other words, we have to figure out what's the largest x and y coordinates for a particular problem. For this problem, it is the point (0, 160) on the y-axis. Hence, the y-axis is limited to $[0, 160]$. If these points are too large (and for some LP problems they are in thousands or millions), do not set any limits and let `matplotlib` do auto scaling. It may not look pretty, but you'll get the general idea of what the lines look like.

A useful tool to use is the function `np.linspace(x, y, z)` below generates a numpy array of `z` equidistant values between `x` and `y`. Here is an example where we generate 10,000 equidistance points between 0 and 160 and then apply the functions `plastic_constraint()` and `steel_constrain()` to each point in `xvals` to generate the corresponding y values for each function. The `xvals`, `yvals1`, and `yvals2`. We will use these arrays for the `matplotlib` plots.

```
import numpy as np
xvals  = np.linspace(0, 160, 10000)
yvals1 = np.array([plastic_constraint(x) for x in xvals])
yvals2 = np.array([steel_constraint(x) for x in xvals])
## x = 0
```

```
x1, y1 = [0, 0], [0, 160]
## y = 0
x2, y2 = [0, 160], [0, 0]
```

We are ready to plot everything with `matplotlib`, label each line, and show the lines on the same plot image. Note that in the code of `plot_teds_constraints()` below, I start the limits of the x- and y-axes below with a negative number. This is done for a better display effect to make the point (0, 0) clearly visible in the graph.

The call `plt.grid()` in `plot_teds_constraints()` shows the grid on the plot's image. If you don't like grids, you can comment it out. The following lines in `plot_teds_constraints()`

```
    plt.plot(xvals, yvals1, label='4x+3y=480', c='red')
    plt.plot(xvals, yvals2, label='3x+6y=720', c='blue')
    plt.plot(x1, y1, label='x=0', c='green')
    plt.plot(x2, y2, label='y=0', c='yellow')
```

show you how you can label and draw curves of different colors. You can check the online `matplotlib` documentation if you want to use customized colors.

```
import matplotlib.pyplot as plt
def plot_teds_constraints():
    def plastic_constraint(x): return -(4/3.0)*x + 160.0
    def steel_constraint(x): return -0.5*x + 120.0
    xvals  = np.linspace(0, 160, 10000)
    yvals1 = np.array([plastic_constraint(x) for x in xvals])
    yvals2 = np.array([steel_constraint(x) for x in xvals])
    fig1   = plt.figure(1)
    fig1.suptitle('Ted\'s Toys Problem')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.ylim([-5, 160])
    plt.xlim([-5, 160])
    x1, y1 = [0, 0], [0, 160]
    x2, y2 = [0, 160], [0, 0]
    plt.grid()
    plt.plot(xvals, yvals1, label='4x+3y=480', c='red')
    plt.plot(xvals, yvals2, label='3x+6y=720', c='blue')
    plt.plot(x1, y1, label='x=0', c='green')
    plt.plot(x2, y2, label='y=0', c='yellow')
    plt.legend(loc='best')
    plt.show()
```

When you call `plot_teds_constraints()` (source code in `cs3430_s24_hw03.py`), you should see the graph shown in Fig. 1. This is what it looks like on Ubuntu 18.04 LTS in Python 3.6.7. The rendering may look slightly different on other platforms. We are lucky in this problem, because for this problem each line intersection point happens to be a corner point of the feasible set. Let us code up all the constraint lines and compute the intersection/corner points.

```
red_line    = (4, 3, 480)
blue_line   = (3, 6, 720)
green_line  = (1, 0, 0)
yellow_line = (0, 1, 0)
cp1 = line_ip(green_line, yellow_line)
cp2 = line_ip(green_line, blue_line)
cp3 = line_ip(blue_line, red_line)
cp4 = line_ip(red_line, yellow_line)
```

All that's left is to define the objective function $5x+4y$ and maximize it on the corner points with `max_obj_fun()`.

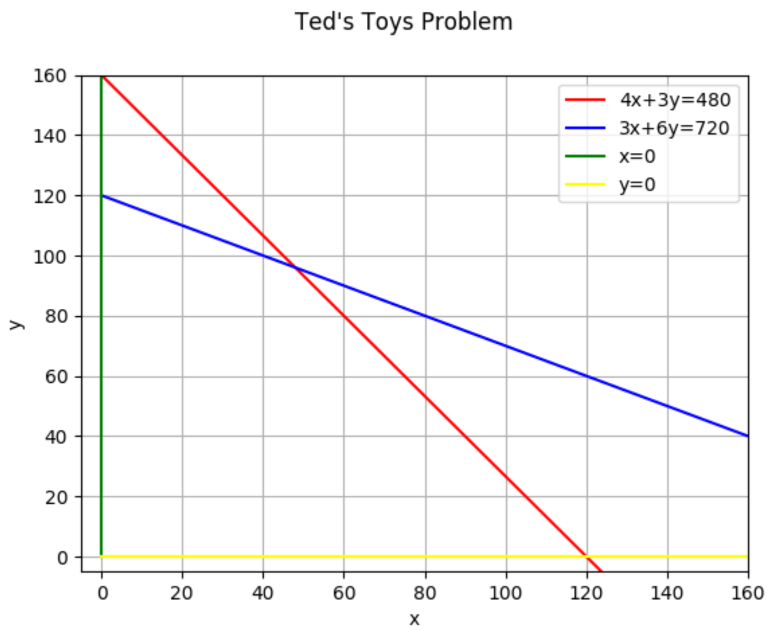

Figure 1: Ted's Toys Problem Constraints.

```
obj_fun = lambda x, y: 5.0*x + 4.0*y
rslt = max_obj_fun(obj_fun, [cp1, cp2, cp3, cp4])
```

And, that is it! Let us put it all together in one function (we will call it `teds_problem()`) and run it. Note that the the function returns the values of x, y, and p, where x and y are the coordinates of the corner point that maximizes the objective function and p (stands for profit) is the value of the objective function at (x, y). In other words, this function returns the number of toy cars (x) and trucks (y) Ted should produce and the maximum profit (p) he will get if (and it's a big if!) he sells them.

```
def teds_problem():
    red_line    = (4, 3, 480)
    blue_line   = (3, 6, 720)
    green_line  = (1, 0, 0)
    yellow_line = (0, 1, 0)
    cp1 = line_ip(green_line, yellow_line)
    cp2 = line_ip(green_line, blue_line)
    cp3 = line_ip(blue_line, red_line)
    cp4 = line_ip(red_line, yellow_line)
    obj_fun = lambda x, y: 5.0*x + 4.0*y
    rslt = max_obj_fun(obj_fun, [cp1, cp2, cp3, cp4])
    ## Let's get the values of x and y out of rslt.
    x = rslt[0][0][0]
    y = rslt[0][1][0]
    p = rslt[1]
    print('num cars   = {}'.format(x))
    print('num trucks = {}'.format(y))
    print('profit     = {}'.format(p))
    return x, y, p

>>> x, y, p = teds_problem()
num cars   = 48.0
num trucks = 96.0
profit     = 624.0
>>> x
48.0
>>> y
```

```
96.0
>>> p
624.0
```

In other words, Ted should make 48 toy cars and 96 toy cars for a profit of 624 dollars.

Use the semi-automatic method detailed in this coding lab to solve Problems 2, 3, and 4 below. For each problem, define one plotting function that plots the constraints (similar to `plot_teds_constraints()` above) and one function that actually solves the problem, prints the solution, and returns the values of x, y, and $p(x, y)$, where p is the objective function and (x, y) is a maximum/minumum corner point (similar to `teds_problem()` above). You should use your implementations of `line_ip()`, `check_line_ip()`, `max_obj_fun()`, and `min_obj_fun()` from Problem 1.

Problem 2: (1 point)

Maximize $p = 3x + y$ subject to

1. $x + y \geq 3$;
2. $3x - y \geq -1$;
3. $x \leq 2$.

Note that feasible set for this problem is bounded. Save your solution to this problem in `plot_problem_2_constraints()` and `problem_2()`.

Problem 3: (1 point)

Maximize $p = x + y$ subject to

1. $x \geq 0$;
2. $y \geq 0$;
3. $x + 2y \geq 6$;
4. $x - y \geq -4$;
5. $2x + y \leq 8$.

Note that the feasible set for this problem is bounded. Save your solution to this problem in `plot_problem_3_constraints()` and `problem_3()`.

Problem 4: (2 points)

A hiker is planning her trail food. The food will include peanuts and raisins. She would like to receive 600 calories and 90 grams of carbohydrates from her food intake. Each gram of raisins contains 0.8 gram of carbohydrates and 3 calories and costs 4 cents. Each gram of peanuts contains 0.2 gram of carbohydrates and 6 calories and costs 5 cents. How much of each ingredient should the hiker take to minimize the cost and to satisfy her dietary constraints?

Save your solution to this problem in `plot_problem_4_constraints()` and `problem_4()`.

What to Submit

Save all your code in `cs3430_s24_hw03.py` and submit it in Canvas. My unit tests are in `cs3430_s24_hw03_uts.py`.

Happy Hacking!