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

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 25, 2020

Learning Objectives

1. Linear Programming in 2D
2. Graphing Linear Constraints
3. Determining Corner Points

Introduction

In this assignment, we'll implement a *semi*-automatic method of solving LP problems in 2D. The method is semi-automatic in the sense that we'll 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 of any number of dimensions. In the meantime, you'll develop better geometric intuitions of how LP works. You'll save your coding solutions in `cs3430_s20_hw03.py` included in the zip and submit it in Canvas.

Problem 1: (2 points)

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 as

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

We can then unpack the coefficients into variables.

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

Let's start this assignment by implementing the function `line_ip(line1, line2)` that takes 2 lines represented as 3-tuples and returns a 2x1 numpy vector `v` that represents their intersection point so that `v[0,0]` is the x-coordinate of the intersection and `v[1,0]` is the y-coordinate of the intersection.

If there is no intersection (i.e., the lines are parallel), the function returns `None`. In implementing this function, use your implementation of `gje()` from Assignment 01 to solve $Ax = b$ for x, where the 2x2 matrix A consists of the coefficients of the two lines and b is a column vector of the 2 C coefficients. Here's a example.

```
>>> from cs3430_s20_hw01 import gje
>>> from cs3430_s20_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.]])
>>> line3 = (4.0, 3.0, 200.0)
>>> line4 = (4.0, 3.0, 250.0)
>>> ip34 = line_ip(line3, line4)
>>> ip34 is None
True
```

Let's 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):
    A1, B1, C1 = line1
    A2, B2, C2 = line2
    x = ip[0, 0]
    y = ip[1, 0]
    assert abs((A1*x + B1*y) - C1) <= err
    assert abs((A2*x + B2*y) - C2) <= err
```

Let's proceed to implement the function `find_line_ips(lines)` that takes an array of lines and returns a list of pairwise intersection points (without `None`'s!) 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 an example.

```
>>> 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])
>>> check_line_ip(line1, line3, ips[1])
>>> check_line_ip(line2, line3, ips[2])
```

Now we need to maximize functions. So, let's 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 2x1 numpy array. The function returns a 2-tuple that consists of a point and the value of f at that point. Here's an example.

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

Problem 2: (2 points)

We have all the machinery in place for solving standard maximum problems (SMP's) in 2D semi-automatically. 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 the corner points.

Let's consider again the Ted's Toys problem we analyzed in Lectures 05 and 06 (see your class notes and handouts for more details). Recall that this problem has the following constraints, where x is the number of cars and y is the number of trucks.

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.

```
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 that we'll plot. 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. You have to figure out what's the largest x and y coordinates for a particular problem. For example, for this problem, it's the point (0, 160) on y axis. Hence, the y axis is limited to $[0, 160]$. If these points are two large, do not set any limits and let `matplotlib` do auto scaling.

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

```
import matplotlib.pyplot as plt
def plot_teds_constraints():
    ### plastic constraint: 4x + 3y <= 480
    def plastic_constraint(x): return -(4/3.0)*x + 160.0
    ### steel constraints: 3x + 6y <= 720
    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])
    ## x = 0
    x1, y1 = [0, 0], [0, 160]
    ## y = 0
    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()`, you should see the graph shown in Fig. 1. We're a bit lucky on this problem, because each line intersection point happens to be a corner point of the feasible set. Let's 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()`.

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

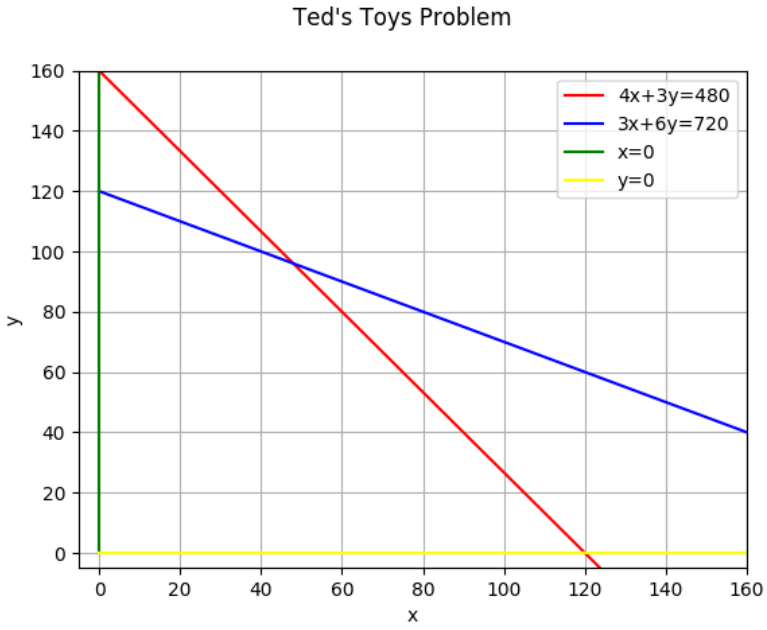

Figure 1: Ted's Problem Constraints.

And, that is it! Let's put it all together in one function, `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 is the value of the objective function at (x, y). In other words, it returns the number of toy cars (x) and trucks (y) Ted has to produce and the maximum profit (p) he'll get 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
```

Use the above methodology to solve the following four problems. In other words, for every problem, define one function that plots the constraints 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 corner point. For problem 2.4, use the decision variables in 100's of units to keep things simple.

Problem 2.1: ($\frac{1}{4}$ point)

Maximize $p = 3x + y$ subject to

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

Save your solution to this problem in `plot_2_1_constraints()` and `problem_2_1()`.

Problem 2.2: ($\frac{1}{4}$ 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. $2x + y \leq 8$.

Save your solution to this problem in `plot_2_2_constraints()` and `problem_2_2()`.

Problem 2.3: ($\frac{1}{4}$ point)

A hiker is planning her trail food. The food will include peanuts and raisins. She'd 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 the dietary constraints?

Save your solution to this problem in `plot_2_3_constraints()` and `problem_2_3()`.

Problem 2.4: ($\frac{5}{4}$ point)

The manager of a paper-box plant is scheduling the work for one production line for a week. The line can produce standard and heavy-duty boxes. Each standard box requires 2 pounds of kraft paper, while each heavy-duty box requires 4 pounds. It takes 8 hours of labor to produce 100 standard boxes and 3 hours of labor to produce 100 heavy-duty boxes. There are 50 tons of kraft paper and 2400 hours of labor available for the week. The manager has a contract to deliver 10,000 heavy-duty boxes by the end of the week. The plant makes a profit of $25.00 per standard box and $50.00 per heavy-duty box. How many standard and heavy-duty boxes should the manager schedule the line to produce?

Save your solution to this problem in `plot_2_4_constraints()` and `problem_2_4()`.

What to Submit

Save all your code in `cs3430_s20_hw03.py` and submit it in Canvas.

Happy Hacking!