assignment4

June 11, 2021

[1]:
```python
# Initialize Otter
import otter
grader = otter.Notebook("assignment4.ipynb")
```

Table of Contents

1 Assignment 4: Modeling and Optimization

Due on June 12 at 11:59 pm

Mathematical modeling of a problem at hand give us a systematic way of finding a solution. For example, a maximum likelihood estimator (assuming it exists), $\hat{\theta}$, is a method for finding the parameter that maximizes the likelihood of the data L_n:

$$L_n(\hat{\theta}; x_1, x_2, \ldots, x_n) = \max_{\theta \in \Theta} L_n(\theta; x_1, x_2, \ldots, x_n)$$

Data $x_1, x_2, \ldots$, set of feasible parameters Θ, and likelihood function L_n are given. We find the parameter that "best" describe the data in the context of the likelihood function.

Many other applications of optimization exists, and this assignment will give a hands-on introduction to a simple linear programming problem.

[2]:
```
import cvxpy as cp
import numpy as np
import pandas as pd
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
```

2 Questions 1-3: Resource Allocation Problem

You are in charge of a company that makes two hot sauces: x_1 liters of Kapatio and x_2 liters of Zriracha. We will use optimization technique to find the "best" manufacturing strategy given our resource constraints.

First, we need to define what we mean by "best" strategy. In this scenario, the goal is to obtain the highest revenue possible. While doing so, there are resource constraints we must satisfy.

For example, in order to manufacture these two hot sauce products, different amount of peppers and vineger are needed. Also, we have only so much total resource available.

Ingridients	Kapatio	Zriracha	Total Available
Pepper	5	7	30
Vineger	4	2	12

2.1 Question 1: Resource Constraints

2.1.1 Question 1.a: Modeling Resource Usage

What is the equation for the amount of pepper needed to manufacture x_1 and x_2. What is the equation for the amount of vinegar? (Use Mathpix to write equations) amount of pepper $= 5x_1 + 7x_2$

amount of vinegar $= 4x_1 + 2x_2$

2.1.2 Question 1.b: Resource Usage vs. Total Resource Constraint

Total amount of pepper needed cannot exceed total available. Write down the inequality expressing this relationship. Do the same for vinegar. These inequalities are your resource constraints. Additinally, variables x_1 and x_2 are non-negative: i.e. amount of manufactured goods cannot be negative.

Rewrite the system of constraint inequalities into a matrix inequality: $Ax \leq b$, where $x = (x_1, x_2)^T$. Arrange rows of A and b such that:

- Row 1: total pepper amount constraint
- Row 2: total vinegar amount constraint
- Row 3: Kapatio non-negativity constraint
- Row 4: Zriracha non-negativity constraint

Less than symbol in $Ax \leq b$ means element-wise.

$$5x_1 + 7x_2 \leq 30$$

$$4x_1 + 2x_2 \leq 12$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

Define matrix `A1` and vector `b1` according to matrix inequality above.

[3]:
```
A1 = np.matrix([[5, 7], [4, 2], [-1,0], [0,-1]])
b1 = np.array([30, 12, 0, 0])
```

[4]:
```
grader.check("q1b2")
```

[4]:
```
q1b2 passed!
```

2.1.3 Visualizing Feasible Region

In a 2-dimensional plot, we will visualize the area that satisfies both of the resource constraints. Draw x_1 on the horizontal axis and x_2 on the vertical axis.

There will be two main components to the plot: * **Lines** indicating constraint boundaries: e.g. the constraint $x_2 \geq 0$ has boundary at $x_2 = 0$. * **Shaded area** indicating feasible regions:

e.g., the whole region $x_2 > 0$ is to be shaded *if* $x_2 \geq 0$ was the only constraint. We will use shading to indicate the region where *all* constraints are satisfied.

[5]:
```
x1_line = np.linspace(-1, 10, 500)
x2_line = np.linspace(-1, 10, 500)
```

2.1.4 Question 1.c: Feasible Region Boundary

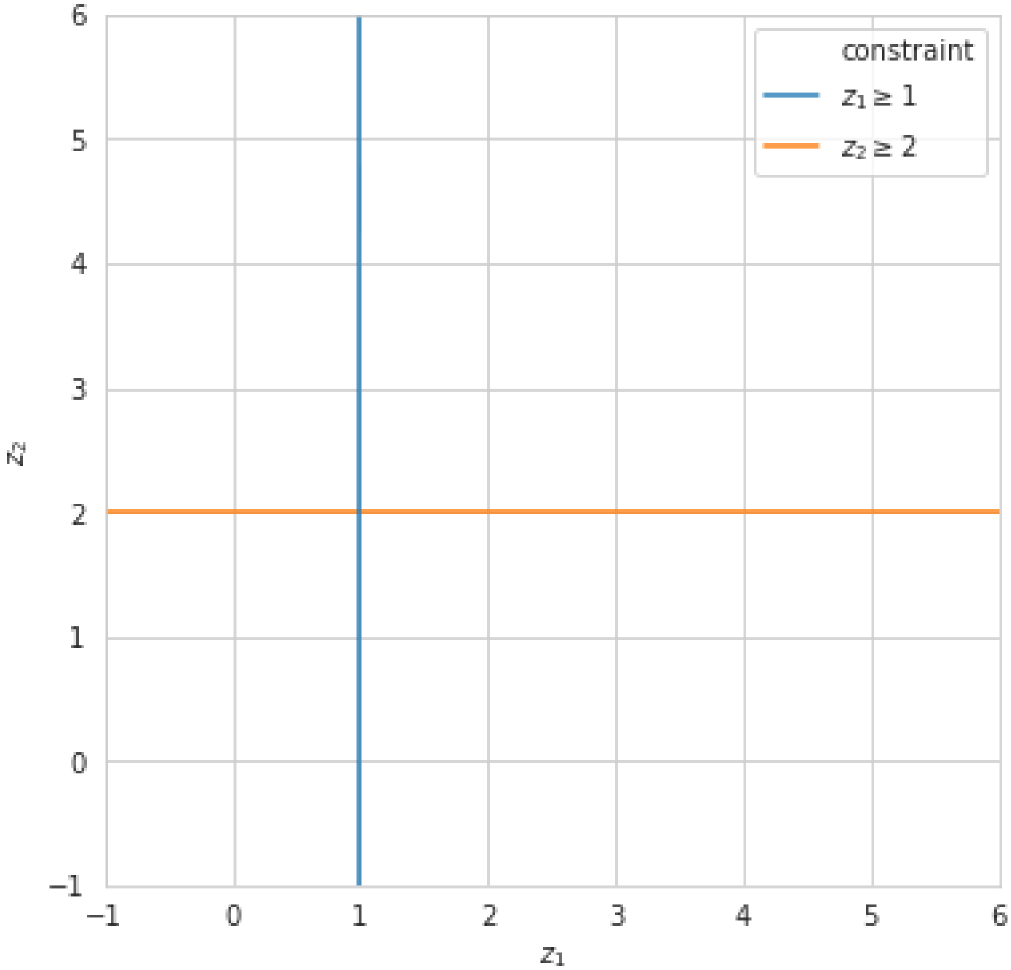
In a list named `boundary`, create four data frames for each equality in $Ax = b$. These lines indicate where the feasible area ends. Set column names as

- `$x_1$`
- `$x_2$`
- `constraints`

Note the use of latex codes (feel free to use Mathpix).

Toy Example: Drawing Boundaries Here is a **toy example** of drawing two constraint boundaries by constructing data frames:

[6]:
```
z1_line = np.linspace(-1, 10, 500)
z2_line = np.linspace(-1, 10, 500)

boundary = [
    pd.DataFrame({
        '$z_1$': np.ones_like(z2_line)*1,
        '$z_2$': z2_line,
        'constraint': '$z_1\geq 1$'
    }),
    pd.DataFrame({
        '$z_1$': z1_line,
        '$z_2$': np.ones_like(z1_line)*2,
        'constraint': '$z_2\geq 2$'
    }),
]
fig, ax = plt.subplots(figsize=(6, 6))
sns.lineplot(x='$z_1$', y='$z_2$', hue='constraint', data=pd.concat(boundary),␣
 ↪ax=ax).axvline(1)
plt.xlim(-1, 6)
plt.ylim(-1, 6)
plt.show()
```



Sometimes, things just do not work as expected.

In the toy example code,

`sns.lineplot(x='$z_1$', y='$z_2$', hue='constraint', data=pd.concat(boundary), ax=ax).axvline(1`

what seems strange about the plotting command? Why was the strange code necessary?

The plotting command seems strange because the .axvline(1) code, from matplotlib, is needed in addition to the .lineplot(), from seaborn, in order to form the intended plot. This strange code is necessary because it allows the $z_1 \geq 1$ constraint to be plotted since .lineplot() is unable to plot the blue vertical constraint line otherwise.

Example: Resource Constraint Boundary Now, create a data frame for the non-negativity constraint $x_2 \geq 0$ as follows:

```
[7]: pd.DataFrame({'$x_1$':x1_line,              ## x_1 can take on any value
                   '$x_2$':x1_line * 0.0,        ## x_2 = 0
```

```
                'constraint':'$x_2 \geq 0$'}), ## constraint equation for labeling
```

```
[7]: (          $x_1$  $x_2$    constraint
 0    -1.000000   -0.0  $x_2 \geq 0$
 1    -0.977956   -0.0  $x_2 \geq 0$
 2    -0.955912   -0.0  $x_2 \geq 0$
 3    -0.933868   -0.0  $x_2 \geq 0$
 4    -0.911824   -0.0  $x_2 \geq 0$
 ..         …      …          …
 495   9.911824    0.0  $x_2 \geq 0$
 496   9.933868    0.0  $x_2 \geq 0$
 497   9.955912    0.0  $x_2 \geq 0$
 498   9.977956    0.0  $x_2 \geq 0$
 499  10.000000    0.0  $x_2 \geq 0$

 [500 rows x 3 columns],)
```

Create a list named `boundary` containing four data frames (each corresponding to a constraint). Concatenate data frames in `boundary` to one data frame named `hull`.

```
[8]: boundary = [
    pd.DataFrame({'$x_1$': 6 - (x2_line * 7/5),
                  '$x_2$': x2_line,
                  'constraint':'$5x_1 + 7x_2 \leq 30$'}),
    pd.DataFrame({'$x_1$': 3 - (x2_line * 0.5),
                  '$x_2$': x2_line,
                  'constraint':'$4x_1 + 2x_2 \leq 12$'}),
    pd.DataFrame({'$x_1$': x2_line * 0.0,
                  '$x_2$': x2_line,
                  'constraint':'$x_1\geq 0$'}),
    pd.DataFrame({'$x_1$': x1_line,
                  '$x_2$': x1_line * 0.0,
                  'constraint':'$x_2\geq 0$'})
]
hull = pd.concat(boundary)
```

```
[9]: grader.check("q1c2")
```

```
[9]: q1c2 passed!
```

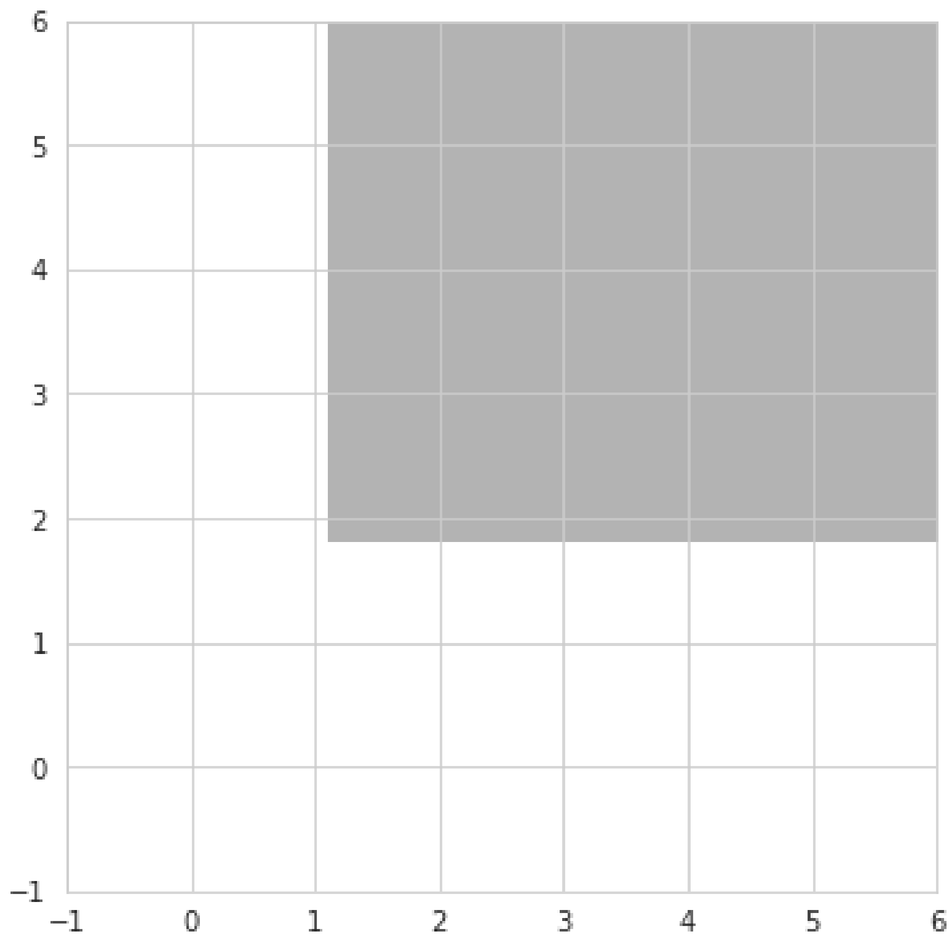
2.1.5 Question 1.d: Interior of Feasible Region

Previous question prepared constraint boundaries, $Ax = b$. In this question, we calculate the interior of the feasible region, which will be shaded in the visualization. First, create a 2-d array of x_1 and x_2 values. If a point (x_1, x_2) satisfies *every* constraint, the point will be colored grey.

For example, in order to shade $\{x_1 : x_1 \geq 1\} \cap \{x_2 : x_2 \geq 2\}$, we can use the `imshow` method.

[10]:

```
z1_line = np.linspace(-1, 6, 10)
z2_line = np.linspace(-1, 6, 10)
z1_grid, z2_grid = np.meshgrid(z1_line, z2_line)

fig, az = plt.subplots(figsize=(6, 6))
az.imshow(
    ((z1_grid >= 1) & (z2_grid >= 2)).astype(int),
    origin='lower',
    extent=(z1_grid.min(), z1_grid.max(), z2_grid.min(), z2_grid.max()),
    cmap="Greys", alpha=0.3, aspect='equal'
)
plt.xlim(-1, 6)
plt.ylim(-1, 6)
plt.show()
```



By dissecting the command below and reading the documentation, report what each of the following lines does:

- `((y1_grid >= 1) & (y2_grid >= 2)).astype(int)` (What is the output of running this command?)

- `origin='lower'`

- `extent=(y1_grid.min(), y1_grid.max(), y2_grid.min(), y2_grid.max())`

- `cmap='Greys'`

- `alpha=0.3`

- `aspect='equal'`

- The command `((y1_grid >= 1) & (y2_grid >= 2)).astype(int)` goes through the different combinations of y1_grid & y2_grid. Then the cobimations are tested with 2 boolean expressions. If the result is True, the statement is coded and displayed as a 1. On the other hand, if the result is False, the statement is coded and displayed as a 0.

- The command `origin='lower'` is a parameter within the .imshow() function in the Matplotlib libary. This command places the [0, 0] index of the array in the lower left corner of the axes.

- The command `extent=(y1_grid.min(), y1_grid.max(), y2_grid.min(), y2_grid.max())` is a parameter within the .imshow() function in the Matplotlib libary. This command defines the bounding box in data coordinates that are given by the tuple containing 4 elements. The 4 different elements are the minimum and maximum values of the y1_grid along with the maximum value of the y_1 grid, the minimum and maximum values of the y2_grid.

- The command `cmap='Greys'` is a parameter within the .imshow() function in the Matplotlib libary. This command gives the shaded region its color of grey by using a pre-defined color name.

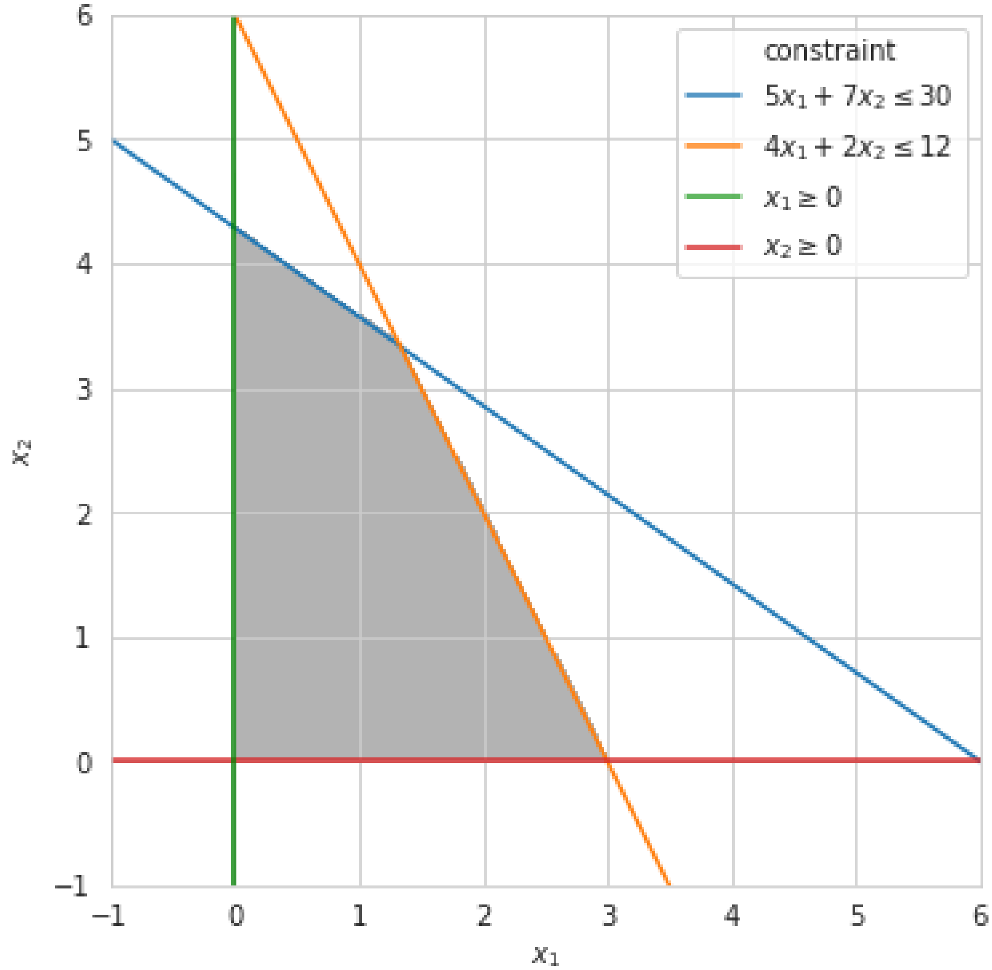
- The command `alpha=0.3` is a parameter within the .imshow() function in the Matplotlib libary. This command defines the opacity of color defined in the cmap parameter for the entire shaded region.

- The command `aspect='equal'` is a parameter used to control the aspect ratio of the axes, which can be chosen from registered keywords of either 'equal' or 'auto'. In this case, 'equal' is used to ensure an aspect ratio of 1 and that the pixels of the image will be square.

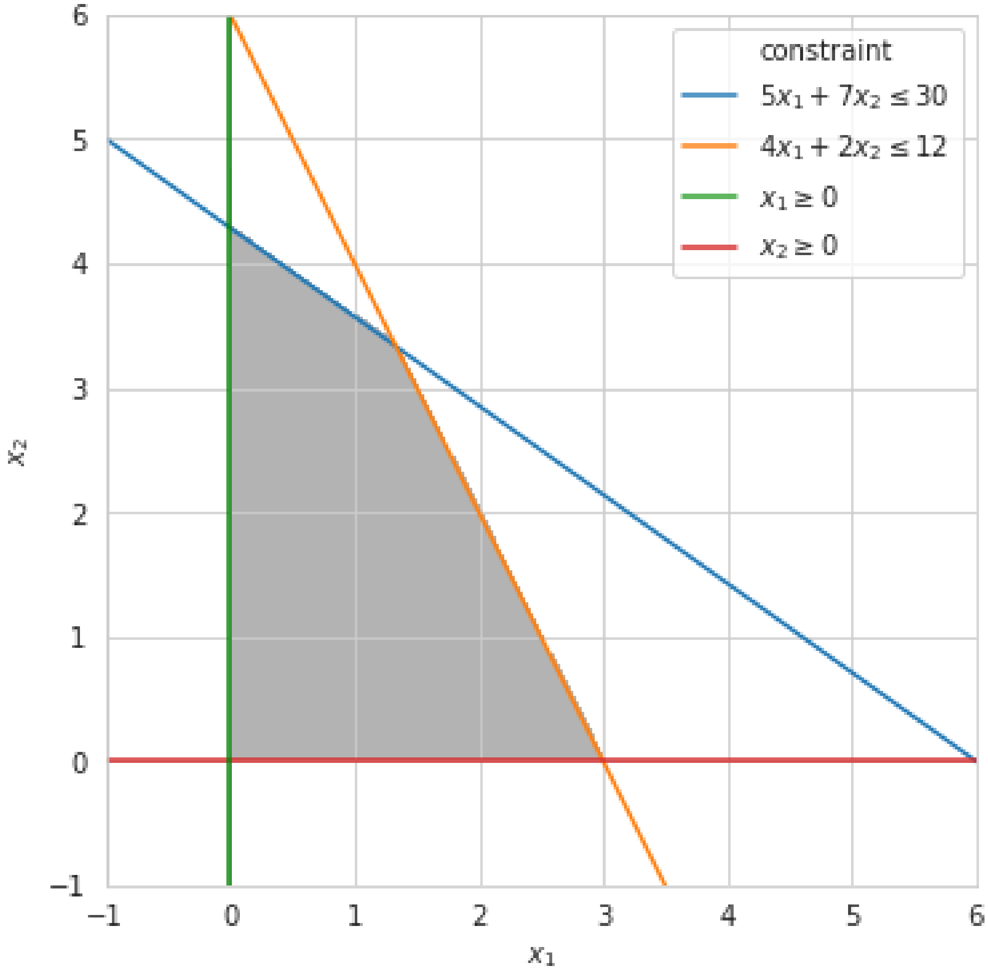
2.1.6 Question 1.e: Visualizing the Feasible Region

Finally, create a figure that shows constraint boundaries and the interior region shaded with a light grey color.

Your output will look like this:



[11]:
```
fig, ax = plt.subplots(figsize=(6, 6))
x1_grid, x2_grid = np.meshgrid(x1_line, x2_line)

ax.imshow(
    ((5*x1_grid + 7*x2_grid <= 30) & (4*x1_grid + 2*x2_grid <= 12) & (x1_grid␣
↪>= 0) & (x2_grid >= 0)).astype(int),
    origin='lower',
    extent=(x1_grid.min(), x1_grid.max(), x2_grid.min(), x2_grid.max()),
    cmap="Greys", alpha = 0.3, aspect='equal')


# ax = sns.lineplot(???).axvline(???)
ax = sns.lineplot(x='$x_1$', y='$x_2$', hue='constraint', data=pd.
↪concat(boundary), ax=ax).axvline(0, color='green')

plt.xlim(-1, 6)
plt.ylim(-1, 6)
plt.show()
```



In the context of linear programming, $Ax \leq b$ is called the *feasible region* (including the appropriate sections of the boundaries). Denote the (shaded) feasible region as set C. Points $(x_1, x_2) \in C$ satisfy all of the constraints.

Describe in plain words the feasible region in the context of hot sauce manufacturing. Specifically, which constraint is violated (if any) by a point at:

- $(x_1, x_2) = (4, 1)$
- $(x_1, x_2) = (0, 5)$

- $(x_1, x_2) = (3, 4)$

In our scenario, the feasible region would be the desired area of producing both hot sauces given resource constraints. For $(x_1, x_2) = (4, 1)$, the constraint that is violated is the production of vinegar as it would require using more than the total supply of vinegar. For $(x_1, x_2) = (0, 5)$, the constraint that is violated is the production of pepper as it would require using more than the total supply of pepper. For $(x_1, x_2) = (3, 4)$, the constraints that are violated is the production of vinegar and pepper as it would require using more than both the total supply of vinegar and pepper.

2.2 Question 2: Objective Function

2.2.1 Question 2.a: Defining Objective Function

Suppose the hot sauces are sold at the same price: \$5 per liter.

What is the equation $f(x)$ for the total revenue as a function of x_1 and x_2?

The function $f(x)$ is called the objective function.

$$f(x) = 5x_1 + 5x_2$$

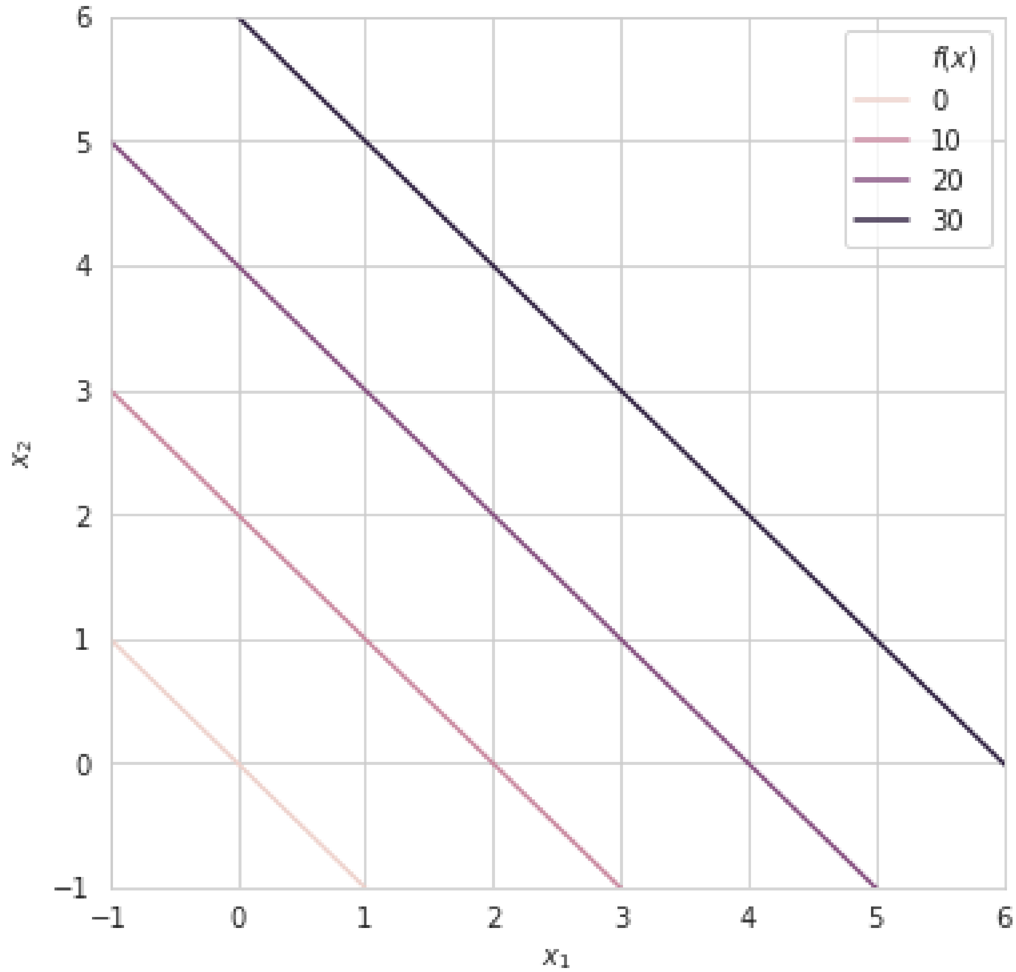
Objective function $f(x)$ is a linear function in x. Therefore, $f(x)$ is a 2-dimesional hyperplane. Note that each value of $f(x)$ defines a line in (x_1, x_2) plane.

For example, $f(x) = 0 = c_1x_1 + c_2x_2$ defines a line. A subspace of equal function value is sometimes referred to as a *level set* or a *contour line* when visualized.

First, create a numpy array of prices `c` for the two hot sauces, x_1 and x_2. Then, create a list `f_vals` containing four data frames of contour lines, $f(x) \in \{0, 10, 20, 30\}$. by creating one data frame for each contour line.

[12]:
```
c = np.array([5,5])

fig, ax = plt.subplots(figsize=(6, 6))
contours = [
    pd.DataFrame({
        '$x_1$': x1_line,
        '$x_2$': (0 - c[0]*x1_line)/c[1],
        '$f(x)$': 0
    }),
    pd.DataFrame({
        '$x_1$': x1_line,
        '$x_2$': (10 - c[0]*x1_line)/c[1],
        '$f(x)$': 10
    }),
    pd.DataFrame({
        '$x_1$': x1_line,
        '$x_2$': (20 - c[0]*x1_line)/c[1],
        '$f(x)$': 20
    }),
    pd.DataFrame({
        '$x_1$': x1_line,
        '$x_2$': (30 - c[0]*x1_line)/c[1],
        '$f(x)$': 30
    })
]
```

```
f_vals = pd.concat(contours)

# ax = sns.lineplot(???)
ax = sns.lineplot(x='$x_1$', y='$x_2$', hue='$f(x)$', data=f_vals, ax=ax)

plt.xlim(-1, 6)
plt.ylim(-1, 6)
plt.show()
```



[13]:
```
grader.check("q2a2")
```

[13]: q2a2 passed!

2.2.2 Question 2.b: Direction of Steepest Increase

Since we want to maximize revenue, we want to increase our objective function as much as possible. Analogous to the minimization example given in a previous lecture, we can repeatedly move in the direction of function increase. In order to determine such direction, compute the gradient of $f(x)$ at $x = (0,0)^T$:

$$\nabla_x f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \end{pmatrix}$$

$$\nabla_x f(x) = \begin{pmatrix} \frac{\partial (5x_1+5x_2)}{\partial x_1} \\ \frac{\partial (5x_1+5x_2)}{\partial x_2} \end{pmatrix}$$

$$\nabla_x f(x) = \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

2.3 Question 3: Putting Pieces Together

2.3.1 Question 3.a: Standard Form of a Linear Programming Problem

Write down the so-called the *standard form* of a linear programming problem:

$$\max_x f(x)$$
$$\text{subject to } Ax \leq b$$

Specifically, write the obejective as an inner product of two vectors: $f(x) = c^T x$, and write the constraint as a vector inequality involving a matrix-vector prduct: $Ax \leq b$, where A is a 4-by-2 matrix.

$$\max c^T x$$
$$\text{subject to } Ax \leq b$$

2.3.2 Question 3.b: Computing the Numerical Solution

Therefore, *maximizing* the revenue is a search over the feasible region for the best point $x^* = (x_1^*, x_2^*)$ that gives the largest revenue. On the otherhand, any *infeasible* point *not* in the feasible region cannot be a solution to the constrained optimization problem.

Notationally, the following expression means the same thing:

$$x^* = \arg \max_{\{x: Ax \leq b\}} f(x)$$

Using CVXPY, solve for the resource allocation problem with constraints.

[14]:
```
# define variables
x = cp.Variable(2)
```

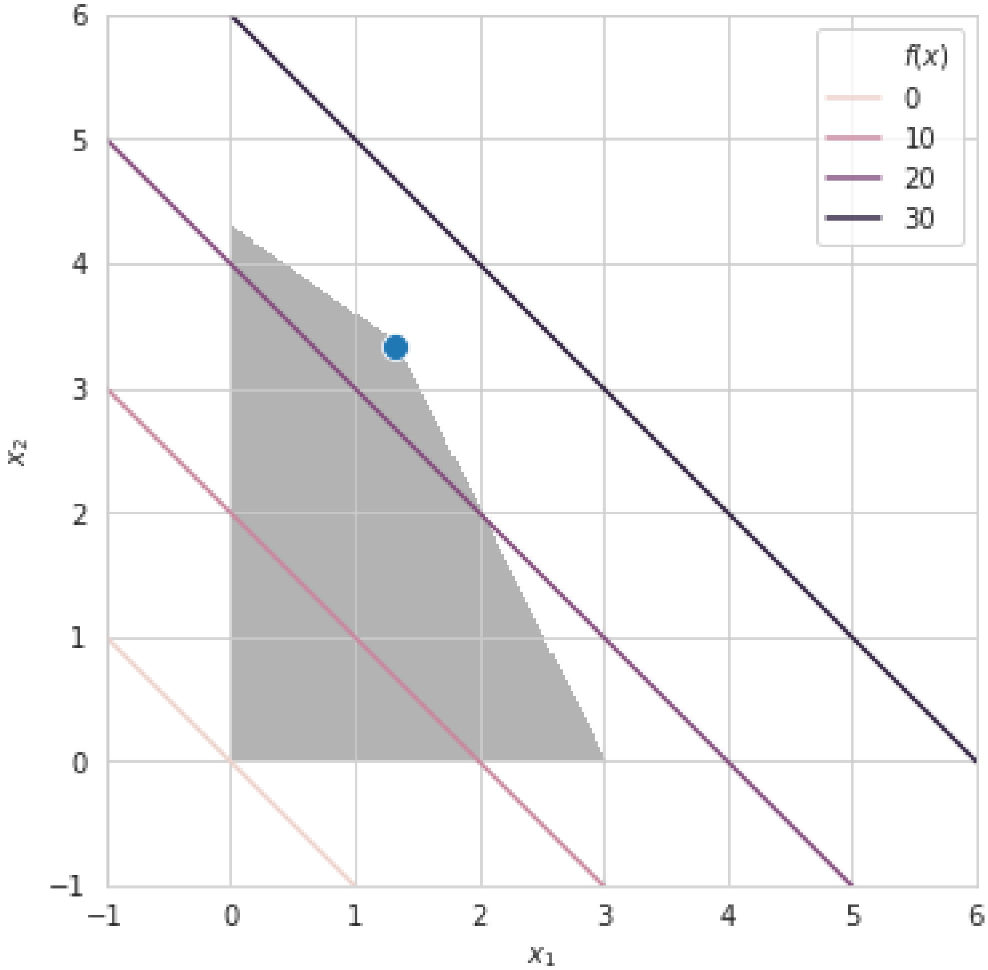
```
# define the linear program
problem = cp.Problem(
    cp.Maximize(c.T@x),
    [A1@x <= b1]
)

fstar1 = problem.solve()                                                    #␣
↪maximum attained function value
xstar1 = pd.DataFrame(x.value.reshape(1, 2), columns=['$x_1$', '$x_2$']) #␣
↪maximizer x for f
```

[15]:
```
grader.check("q3b")
```

[15]: q3b passed!

2.3.3 Question 3.c: Plotting the optimal solution

[16]:
```
fig, ax = plt.subplots(figsize=(6, 6))
x1_grid, x2_grid = np.meshgrid(x1_line, x2_line)
ax.imshow(
    (
        (A1[0,0]*x1_grid + A1[0,1]*x2_grid <= b1[0]) & # Pepper constraints
        (A1[1,0]*x1_grid + A1[1,1]*x2_grid <= b1[1]) & # Vinegar constraints
        (A1[2,0]*x1_grid + A1[2,1]*x2_grid <= b1[2]) &
        (A1[3,0]*x1_grid + A1[3,1]*x2_grid <= b1[3])   # non-negativity␣
↪constraints
    ),
    origin='lower',
    extent=(x1_grid.min(), x1_grid.max(), x2_grid.min(), x2_grid.max()),
    cmap="Greys", alpha = 0.3, aspect='equal'
)
sns.scatterplot(x='$x_1$', y='$x_2$', data=xstar1, ax=ax, s=100)
sns.lineplot(x='$x_1$', y='$x_2$', hue='$f(x)$', data=f_vals, ax=ax)
plt.xlim(-1, 6)
plt.ylim(-1, 6)
plt.show()
```



3 Question 4: Nutrition Problem

During the second world war, the US Army set out to save money without damaging the nutritional health of members of the armed forces.

> According to this source, the following problem is a *simple variation of the well-known diet problem that was posed by George Stigler and George Dantzig: how to choose foods that satisfy nutritional requirements while minimizing costs or maximizing satiety.*
>
> *Stigler solved his model "by hand" because technology at the time did not yet support more sophisticated methods. However, in 1947, Jack Laderman, of the US National Bureau of Standards, applied the simplex method (an algorithm that was recently proposed by George Dantzig) to Stigler's model. Laderman and his team of nine linear programmers, working on desk calculators, showed that Stigler's heuristic approximation was very close to optimal (only 24 cents per year over the optimum found by the simplex method)*

and thus demonstrated the practicality of the simplex method on large-scale, real-world problems.

The problem that is solved in this example is to minimize the cost of a diet that satisfies certain nutritional constraints.

The file `foods.csv` contains calorie, nutritional content, serving size, and price per serving information about 64 foods. Read it into a data frame named `foods`.

[17]:
```
foods = pd.read_csv('files/foods.csv')
print(foods)
```

```
                          Name  Calories  Cholesterol  Total_Fat  Sodium  \
0              Frozen Broccoli      73.8          0.0        0.8    68.2
1                 Carrots, Raw      23.7          0.0        0.1    19.2
2                  Celery, Raw       6.4          0.0        0.1    34.8
3                  Frozen Corn      72.2          0.0        0.6     2.5
4          Lettuce, Iceberg,Raw      2.6          0.0        0.0     1.8
..                         …         …           …         …       …
59            New Eng Clam Chwd     175.7         10.0        5.0  1864.9
60                 Tomato Soup     170.7          0.0        3.8  1744.4
61    New Eng Clam Chwd, w/Mlk     163.7         22.3        6.6   992.0
62       Crm Mshrm Soup, w/Mlk     203.4         19.8       13.6  1076.3
63     Bean Bacon Soup, w/Watr     172.0          2.5        5.9   951.3

    Carbohydrates  Dietary_Fiber  Protein    Vit_A  Vit_C  Calcium  Iron  \
0            13.6            8.5      8.0   5867.4  160.2    159.0   2.3
1             5.6            1.6      0.6  15471.0    5.1     14.9   0.3
2             1.5            0.7      0.3     53.6    2.8     16.0   0.2
3            17.1            2.0      2.5    106.6    5.2      3.3   0.3
4             0.4            0.3      0.2     66.0    0.8      3.8   0.1
..            …             …       …       …      …      …    …
59           21.8            1.5     10.9     20.1    4.8     82.8   2.8
60           33.2            1.0      4.1   1393.0  133.0     27.6   3.5
61           16.6            1.5      9.5    163.7    3.5    186.0   1.5
62           15.0            0.5      6.1    153.8    2.2    178.6   0.6
63           22.8            8.6      7.9    888.0    1.5     81.0   2.0

              Serving  Price/Serving ($)
0          10 Oz Pkg               0.16
1   1/2 Cup Shredded               0.07
2             1 Stalk               0.04
3             1/2 Cup               0.18
4              1 Leaf               0.02
..                …                  …
59      1 C (8 Fl Oz)               0.75
60      1 C (8 Fl Oz)               0.39
61      1 C (8 Fl Oz)               0.99
62      1 C (8 Fl Oz)               0.65
```

```
63     1 C (8 Fl Oz)              0.67

[64 rows x 14 columns]
```

The file `nutritional_constraints.csv` contains healthy nutritional range constraints. Minimum and maximum allowed nutritional contents can be found in this file. Name the variable `requirements`.

```
[18]: requirements = pd.read_csv('files/nutritional_constraints.csv')
print(requirements)
```

```
            Name Unit   Min    Max
0       Calories  cal  2000   2250
1    Cholesterol   mg     0    300
2      Total_Fat    g     0     65
3         Sodium   mg     0   2400
4   Carbohydrates   g     0    300
5   Dietary_Fiber   g    25    100
6        Protein    g    50    100
7          Vit_A   IU  5000  50000
8          Vit_C   IU    50  20000
9        Calcium   mg   800   1600
10          Iron   mg    10     30
```

Extract the nutritional content of foods into a 2-d array named `ncontent`.

```
[19]: ncontent = foods.iloc[:, range(1,12)].T
print(ncontent)
```

```
                    0        1     2      3     4      5      6      7      8  \
Calories         73.8     23.7   6.4   72.2   2.6   20.0  171.5   88.2  277.4
Cholesterol       0.0      0.0   0.0    0.0   0.0    0.0    0.0    0.0  129.9
Total_Fat         0.8      0.1   0.1    0.6   0.0    0.1    0.2    5.5   10.8
Sodium           68.2     19.2  34.8    2.5   1.8    1.5   15.2    8.1  125.6
Carbohydrates    13.6      5.6   1.5   17.1   0.4    4.8   39.9    2.2    0.0
Dietary_Fiber     8.5      1.6   0.7    2.0   0.3    1.3    3.2    1.4    0.0
Protein           8.0      0.6   0.3    2.5   0.2    0.7    3.7    9.4   42.2
Vit_A          5867.4  15471.0  53.6  106.6  66.0  467.7    0.0   98.6   77.4
Vit_C           160.2      5.1   2.8    5.2   0.8   66.1   15.6    0.1    0.0
Calcium         159.0     14.9  16.0    3.3   3.8    6.7   22.7  121.8   21.9
Iron              2.3      0.3   0.2    0.3   0.1    0.3    4.3    6.2    1.8

                    9  …     54     55      56     57      58      59  \
Calories        358.2  …  108.0  142.0   150.1  184.8   158.1   175.7
Cholesterol       0.0  …    0.0    0.0    12.3    7.2    10.0    10.0
Total_Fat        12.3  …    1.0    7.4     4.6    4.0     3.8     5.0
Sodium         1237.1  …  486.2  149.7  1862.2  964.8  1915.1  1864.9
Carbohydrates    58.3  …   22.5   17.8    18.7   26.8    20.4    21.8
Dietary_Fiber    11.6  …    0.9    1.8     1.5    4.1     4.0     1.5
```

```
Protein             8.2  …    2.6    2.0     7.9    11.1    11.2    10.9
Vit_A            3055.2  …    0.0   55.6  1308.7  4872.0  3785.1    20.1
Vit_C              27.9  …    0.0    0.0     0.0     7.0     4.8     4.8
Calcium            80.2  …   10.2   43.7    27.1    33.6    32.6    82.8
Iron                2.3  …    1.2    0.4     1.5     2.1     2.2     2.8

                    60     61      62     63
Calories         170.7  163.7   203.4  172.0
Cholesterol        0.0   22.3    19.8    2.5
Total_Fat          3.8    6.6    13.6    5.9
Sodium          1744.4  992.0  1076.3  951.3
Carbohydrates     33.2   16.6    15.0   22.8
Dietary_Fiber      1.0    1.5     0.5    8.6
Protein            4.1    9.5     6.1    7.9
Vit_A           1393.0  163.7   153.8  888.0
Vit_C            133.0    3.5     2.2    1.5
Calcium           27.6  186.0   178.6   81.0
Iron               3.5    1.5     0.6    2.0

[11 rows x 64 columns]
```

3.0.1 Question 4.a: Define Constraints

To avoid eating the same foods, limit each food intake to be 2 or less. Also, one cannot consume less than zero servings. Furthermore, apply the nutritional constraints as specified in `nutritional_constraints.csv` (assume that the units are the same as food nutritional contents)

Note that a range constraints, e.g., $2000 \leq$ total calories ≤ 2250, can be written as two constraints: total calories ≤ 2250 and $-$total calories ≤ -2000. Hence, we can rewrite caloric intake constraints as

$$-(\text{calories in frozen broccoli})x_0 - (\text{calories in raw carrots})x_1 - \cdots - (\text{calories in bean bacon soup, w/watr})x_{63} =$$

$$\text{calories in frozen broccoli})x_0 + (\text{calories in raw carrots})x_1 + \cdots + (\text{calories in bean bacon soup, w/watr})x_{63} =$$

where vector c contains calorie information for all 64 foods and x containts servings consumed of each food. Matrix U and vector w would be such that

$$U = \begin{pmatrix} -c^T \\ c^T \end{pmatrix} \text{ and } w = \begin{pmatrix} -2000 \\ 2250 \end{pmatrix},$$

and the matrix-vector inequality would be $Ux \leq w$. Range constraints of each food can be implemented similarly with identity matrices.

Denote nutritional content information from `foods` data frame as A and denote the `Min` and `Max` columns of `requirements` as vector b_L and b_U, respectively. Construct M and d in $Mx \leq d$ using I (identity matrix), A, b_L, b_U, and other constants, so that all the range constraints are expressed in $Mx \leq d$. (This is a theory question. No coding is involved)

$$M_{150,64} = \begin{pmatrix} -I_{64,64} \\ I_{64,64} \\ -A_{11,64} \\ A_{11,64} \end{pmatrix}$$

$$d_{150,1} = \begin{pmatrix} 0_{64,1} \\ 2_{64,1} \\ -b_{L_{11,1}} \\ b_{U_{11,1}} \end{pmatrix}$$

3.0.2 Question 4.b: Create Python Variables

Denote the servings of each food as x_i where i is the row index of each food in `foods` data frame: i.e. x_0 indicates number of servings of frozen broccoli, x_1 indicates that of raw carrots, etc.

- Create cost vector `cost` that gives per serving cost.
- Create matrix `M` and `d` that lists nutritional content in the following order:
  - Non-negativity constraint of food consumed: i.e. 0 servings or more
  - Upper limit on food consumed: i.e. 2 servings or less
  - Lower limit on consumption of each nutrition: i.e. following `Min` column
  - Upper limit on consumption of each nutrition: i.e. following `Max` column

[20]:
```
M = np.concatenate([
    -np.eye(64),
    np.eye(64),
    -ncontent,
    ncontent
])

d = np.concatenate([
    np.zeros((64)),
    2*np.ones((64)),
    -requirements['Min'],
    requirements['Max'],
])

cost = foods['Price/Serving ($)'].values
```

[21]:
```
grader.check("q4b")
```

[21]: q4b passed!

3.0.3 Question 4.c: Solve the Problem

Create cvxpy variable `servings` to represent the number of servings of food, and use cvxpy to solve for the optimal solution.

Choose ECOS as your solver.

[22]:
```
servings = cp.Variable(64)

# define the linear program
nutrition_problem = cp.Problem(
    cp.Minimize(cost.T@servings),
    [M@servings <= d]
)

# solve linear programming problem with ECOS solver
# https://www.cvxpy.org/tutorial/advanced/index.html?
 ↪highlight=osqp#choosing-a-solver
fstar2 = nutrition_problem.solve(solver = 'ECOS')
xstar2 = servings.value
```

[23]:
```
grader.check("q4c")
```

[23]: q4c passed!

3.0.4 Question 4.d: Interpreting the Results

State the results in the context of the problem. How much of each food was consumed? List the foods and their calculated amounts. What is the total cost of feeding one soldier?

[24]:
```
result = pd.DataFrame({'Name': foods ['Name'], 'Servings': xstar2})
total_cost = xstar2 * foods['Price/Serving ($)']
total_cost.sum()
```

[24]: 1.2484722937208912

[25]:
```
result
```

[25]:

```
                        Name      Servings
0            Frozen Broccoli  8.024228e-02
1               Carrots, Raw  2.240289e-01
2                Celery, Raw  1.401558e-11
3                Frozen Corn  3.148633e-12
4        Lettuce, Iceberg,Raw  3.018374e-11
..                       ...           ...
59          New Eng Clam Chwd  6.674082e-13
60                Tomato Soup  1.923350e-12
61  New Eng Clam Chwd, w/Mlk  3.453333e-13
62        Crm Mshrm Soup, w/Mlk  7.663001e-13
63      Bean Bacon Soup, w/Watr  7.326201e-13

[64 rows x 2 columns]
```

The total cost of feeding 1 soilder is about $1.25. Furthermore, the chart provides a list of the foods and their calculated amounts/servings consumed.

Cell intentionally blank

To double-check your work, the cell below will rerun all of the autograder tests.

[26]:
```
grader.check_all()
```

[26]:
```
q1b2 passed!

q1c2 passed!

q2a2 passed!

q3b passed!

q4b passed!

q4c passed!
```

3.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[27]:
```
# Save your notebook first, then run this cell to export your submission.
grader.export()
```

```
<IPython.core.display.HTML object>
```