DS-210: Programming for Data Science

Lecture 8: Optimization and Linear Programming

Updates

- HW 1 grades to be posted today
- HW 2 due today
- HW 3 already posted yesterday

Optimization

- Traditional area of **Operations Research**

Optimization

- Traditional area of **Operations Research**

Why study optimization?

- Important when making business decision based on data
- Also at heart of many machine learning methods
    - k–means (previous lecture)
    - deep learning and neural networks

Today: Linear Programming

General setting

- a number of real variables: $x_1, x_2, \ldots, x_n$
- a number of linear constraints:

$$b_{1,1}x_1 + \cdots + b_{1,n}x_n \leq d_1$$

$$\vdots$$

$$b_{t,1}x_1 + \cdots + b_{t,n}x_n \leq d_t$$

- Goal: maximize or minimize
  $c_1x_1 + c_2x_2 + \cdots + c_nx_n$

Today: Linear Programming

General setting

- a number of real variables: $x_1, x_2, \ldots, x_n$
- a number of linear constraints:

$$b_{1,1}x_1 + \cdots + b_{1,n}x_n \leq d_1$$

$$\vdots$$

$$b_{t,1}x_1 + \cdots + b_{t,n}x_n \leq d_t$$

- Goal: maximize or minimize

$$c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

Simple Example

- variables: x and y
- linear constraints:

$$x \geq 0$$
$$y \geq 0$$
$$5x + 4y \leq 20$$
$$3x + 5y \leq 15$$

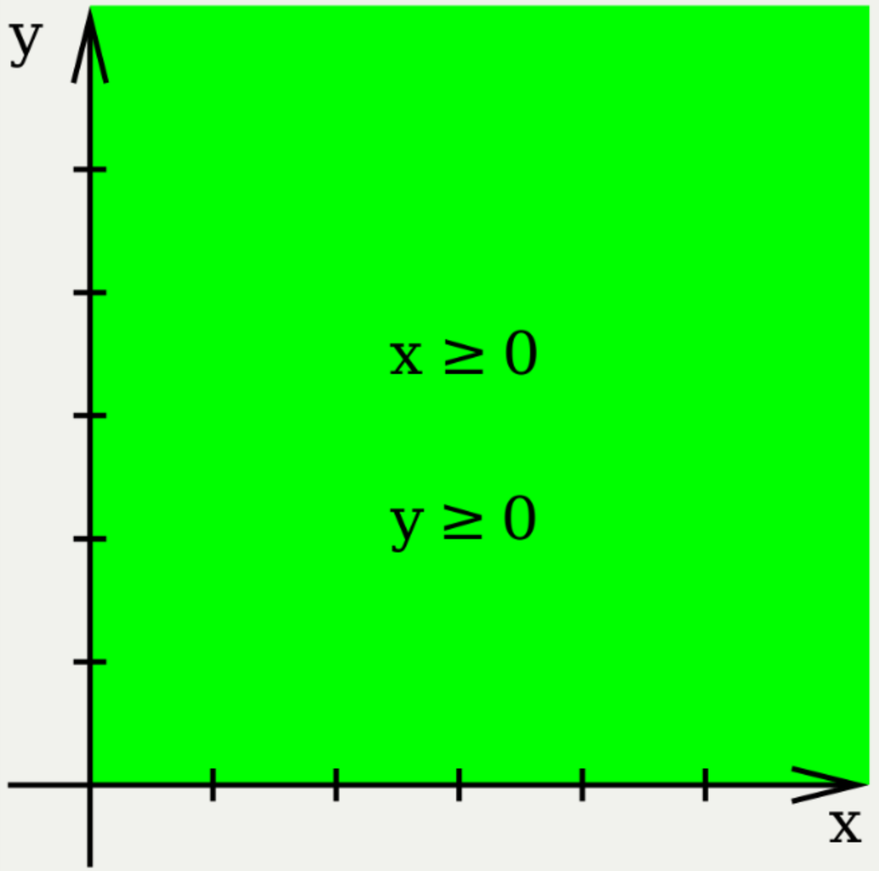
- **Goal:** maximize $x + y$

Simple Example

- variables: x and y
- linear constraints:

$$x \geq 0$$
$$y \geq 0$$
$$5x + 4y \leq 20$$
$$3x + 5y \leq 15$$

- **Goal:** maximize $x + y$

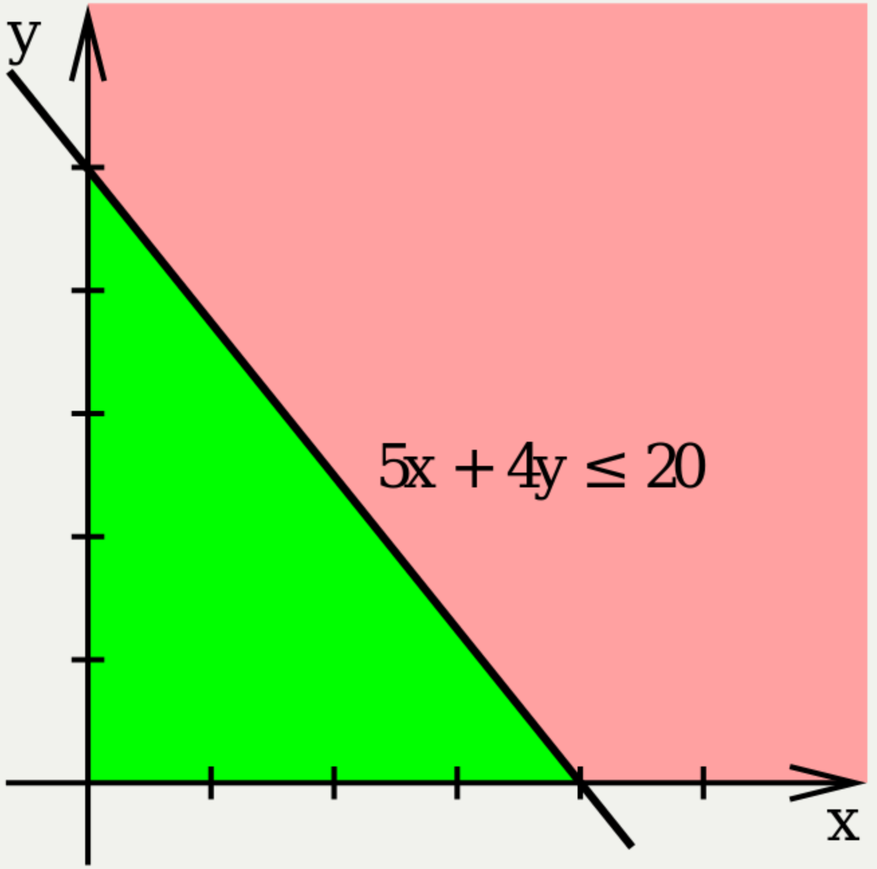


Simple Example

- variables: x and y
- linear constraints:

$$x \geq 0$$
$$y \geq 0$$
$$5x + 4y \leq 20$$
$$3x + 5y \leq 15$$

- **Goal:** maximize $x + y$

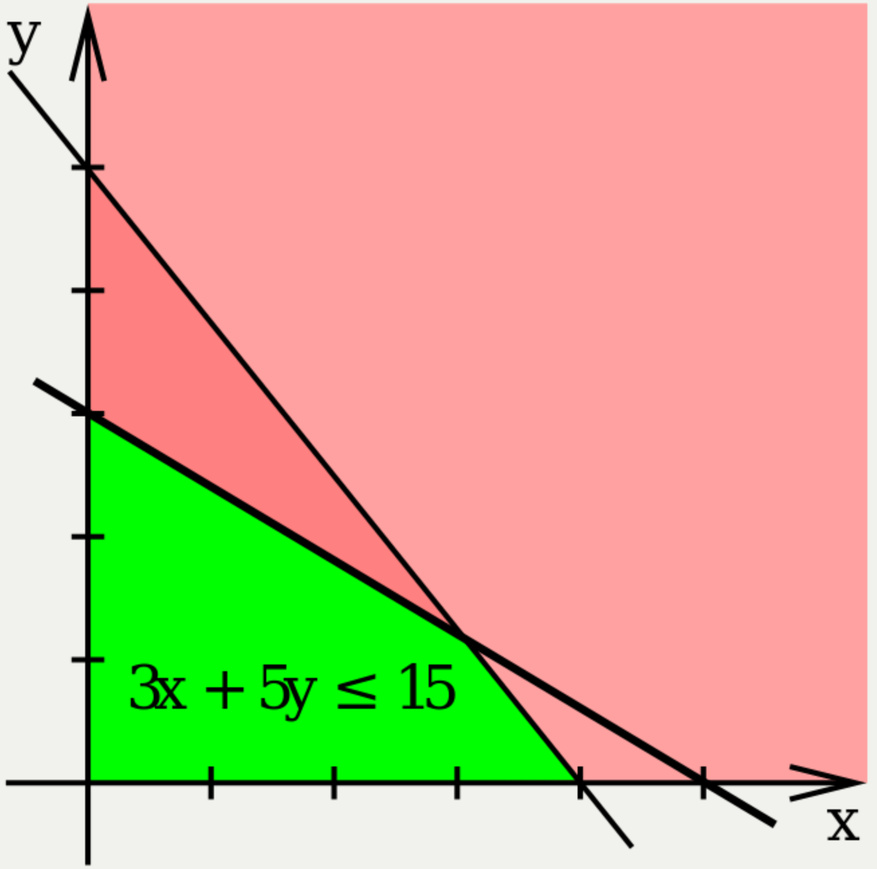


Simple Example

- variables: x and y
- linear constraints:

$$x \geq 0$$
$$y \geq 0$$
$$5x + 4y \leq 20$$
$$3x + 5y \leq 15$$

- **Goal:** maximize $x + y$

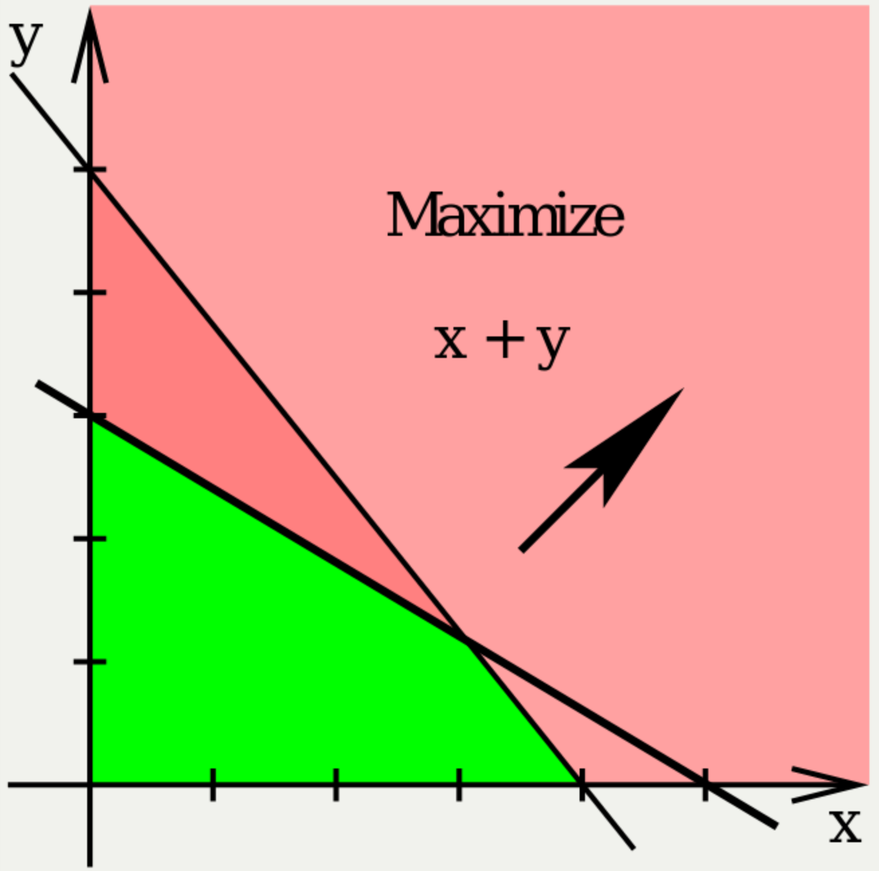


Simple Example

- variables: x and y
- linear constraints:

$$x \geq 0$$
$$y \geq 0$$
$$5x + 4y \leq 20$$
$$3x + 5y \leq 15$$

- **Goal:** maximize $x + y$

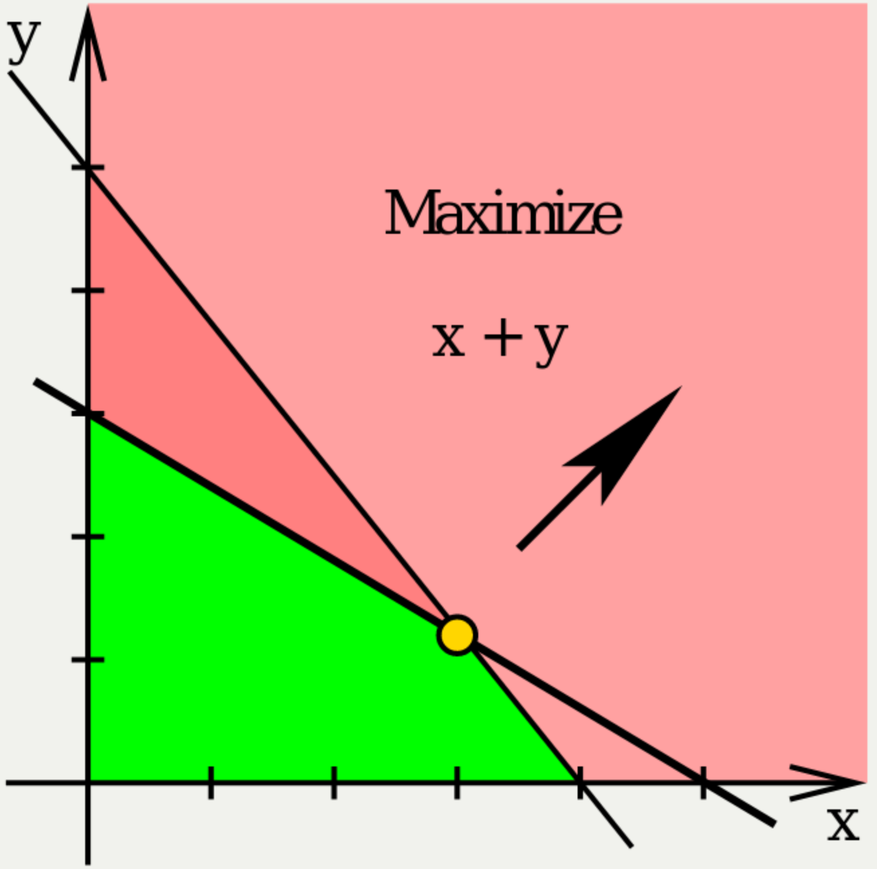


Simple Example

- variables: x and y
- linear constraints:

$$x \geq 0$$
$$y \geq 0$$
$$5x + 4y \leq 20$$
$$3x + 5y \leq 15$$

- **Goal:** maximize $x + y$



Example: get cars to dealerships cheaply

- Five dealerships with demands:

  `[20,40,10,25,10]`

- Three warehouses with inventory:

  `[25,45,40]`

- The cost of moving **one** car from warehouse `i` to dealership `j`: `cost[i][j]`

Goal: Satisfy all demands while paying as little as possible for moving cars

In [1]:
```
demands = [20,40,10,25,10]
inventory = [25,45,40]
```

In [2]:
```
import numpy as np
# set costs to random between 1000 and 2000
cost = np.random.uniform(low=1000.0,high=2000.0,
                         size=(3,5))
print(cost)
```

```
[[1485.41300256 1838.31428107 1324.37968618 1577.65320408 1672.03912278]
 [1366.19702012 1056.23740292 1218.76659577 1912.66843972 1931.46773677]
 [1088.56968831 1974.23851271 1377.95928388 1883.36675632 1780.14736407]]
```

Example: get cars to dealerships cheaply

- Five dealerships with demands:

  `[20,40,10,25,10]`

- Three warehouses with inventory:

  `[25,45,40]`

- The cost of moving **one** car from warehouse `i` to dealership `j`: `cost[i][j]`

Goal: Satisfy all demands while paying as little as possible for moving cars

```
In [1]: demands = [20,40,10,25,10]
        inventory = [25,45,40]
```

```
In [2]: import numpy as np
        # set costs to random between 1000 and 2000
        cost = np.random.uniform(low=1000.0,high=2000.0,
                                 size=(3,5))
        print(cost)
```

```
[[1485.41300256 1838.31428107 1324.37968618 1577.6532040
8 1672.03912278]
 [1366.19702012 1056.23740292 1218.76659577 1912.6684397
2 1931.46773677]
 [1088.56968831 1974.23851271 1377.95928388 1883.3667563
2 1780.14736407]]
```

Variables $x_{i,j}$**:** correspond to the number of cars moved from warehouse i to dealership j.

What constraints do we need?

Example: get cars to dealerships cheaply

Can't move negative numbers of cars:

$$x_{i,j} \geq 0$$

Example: get cars to dealerships cheaply

Can't move negative numbers of cars:

$$x_{i,j} \geq 0$$

Each warehouse i can't send more than its inventory:

$$\sum_{j=1}^{5} x_{i,j} \leq \text{inventory}[i]$$

Example: get cars to dealerships cheaply

Can't move negative numbers of cars:

$$x_{i,j} \geq 0$$

Each warehouse i can't send more than its inventory:

$$\sum_{j=1}^{5} x_{i,j} \leq \text{inventory}[i]$$

Demand of each dealership j is satisfied:

$$\sum_{i=1}^{3} x_{i,j} = \text{demand}[j]$$

Example: get cars to dealerships cheaply

Can't move negative numbers of cars:

$$x_{i,j} \geq 0$$

Each warehouse i can't send more than its inventory:

$$\sum_{j=1}^{5} x_{i,j} \leq \text{inventory}[i]$$

Demand of each dealership j is satisfied:

$$\sum_{i=1}^{3} x_{i,j} = \text{demand}[j]$$

Goal: minimize total cost $\sum_{i=1}^{3} \sum_{j=1}^{5} x_{i,j} \cdot \text{cost}[i][j]$

Solving linear programming with SciPy

https://docs.scipy.org/doc/scipy/reference/generated/scipy.organize.linprog.html

scipy.optimize.linprog

scipy.optimize.linprog(*c, A_ub=None, b_ub=None, A_eq=None, b_eq=None, bounds=None, method='interior-point', callback=None, options=None, x0=None*) [source]

Linear programming: minimize a linear objective function subject to linear equality and inequality constraints.

Linear programming solves problems of the following 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 x is a vector of decision variables; c, b_{ub}, b_{eq}, l, and u are vectors; and A_{ub} and A_{eq} are matrices.

Alternatively, that's:

minimize:

```
c @ x
```

such that:

```
A_ub @ x <= b_ub
A_eq @ x == b_eq
lb <= x <= ub
```

Note that by default `lb = 0` and `ub = None` unless specified with `bounds`.

Notes:

- Make sure you understand the matrix multiplication notation in the docs
- Adjust your linear program to the required form:
  - Make constraints "less then" with variables on the left
  - Make minimization the goal (multiply by -1 if you are maximizing!)

Applying SciPy to our problem

In [3]:
```
# convert cost to one dimensional vector
c = cost.ravel()
print(c)
```

```
[1485.41300256 1838.31428107 1324.37968618 1577.65320408 1672.03912278
 1366.19702012 1056.23740292 1218.76659577 1912.66843972 1931.46773677
 1088.56968831 1974.23851271 1377.95928388 1883.36675632 1780.14736407]
```

Applying SciPy to our problem

In [3]:
```
# convert cost to one dimensional vector
c = cost.ravel()
print(c)
```

```
[1485.41300256 1838.31428107 1324.37968618 1577.65320408 1672.03912278
 1366.19702012 1056.23740292 1218.76659577 1912.66843972 1931.46773677
 1088.56968831 1974.23851271 1377.95928388 1883.36675632 1780.14736407]
```

In [4]:
```
# make sure not too much transported out of each warehouse
zeros,ones = [0] * 5, [1] * 5
A_ub = [
    ones + zeros + zeros,
    zeros + ones + zeros,
    zeros + zeros + ones
]
b_ub = inventory
A_ub,b_ub
```

Out[4]:
```
([[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]],
 [25, 45, 40])
```

In [5]:
```
# make sure demands at dealerships are satisified
def dealearship(j):
    jth = [1 if k == j else 0 for k in range(5)]
    return jth + jth + jth
A_eq = [dealearship(j) for j in range(5)]
b_eq = demands
A_eq,b_eq
```

Out[5]:
```
([[1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
  [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0],
  [0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]],
 [20, 40, 10, 25, 10])
```

In [5]:
```
# make sure demands at dealerships are satisified
def dealearship(j):
    jth = [1 if k == j else 0 for k in range(5)]
    return jth + jth + jth
A_eq = [dealearship(j) for j in range(5)]
b_eq = demands
A_eq,b_eq
```

Out[5]:
```
([[1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
  [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
  [0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
  [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0],
  [0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1]],
 [20, 40, 10, 25, 10])
```

In [6]:
```
# make sure variables are non-negative
# (None means no upper or lower bound)
bounds = [(0,None) for _ in range(15)]
```

In [7]:
```
from scipy.optimize import linprog
linprog(c,A_ub,b_ub,A_eq,b_eq,bounds,method='revised simplex')
```

Out[7]:
```
     con: array([0., 0., 0., 0., 0.])
     fun: 134247.32302369346
 message: 'Optimization terminated successfully.'
     nit: 14
   slack: array([0., 0., 5.])
  status: 0
 success: True
       x: array([ 0.,  0.,  0., 25.,  0.,  0., 40.,  5.,  0.,  0., 20.,  0.,  5.,
        0., 10.])
```

Various optimization options

Parameter `method`

- Simplex (`simplex`, `revised simplex`)
    - better for this class
    - will give you "true" zeros and more accurate numbers
- Default method: Interior point (`interior-point`)
    - faster for some inputs
- Look into `highs-*` for more highly optimized versions

Various optimization options

Parameter `method`

- Simplex (`simplex`, `revised simplex`)
  - better for this class
  - will give you "true" zeros and more accurate numbers
- Default method: Interior point (`interior-point`)
  - faster for some inputs
- Look into `highs-*` for more highly optimized versions

Digression: How to check if something is zero?

Various optimization options

Parameter `method`

- Simplex (`simplex`, `revised simplex`)
  - better for this class
  - will give you "true" zeros and more accurate numbers
- Default method: Interior point (`interior-point`)
  - faster for some inputs
- Look into `highs-*` for more highly optimized versions

Digression: How to check if something is zero?

With finite–precision computer arithmetic, you should check if $|x| < 10^{-6}$ or another sufficiently small number

Various optimization options

Parameter `method`

- Simplex (`simplex`, `revised simplex`)
  - better for this class
  - will give you "true" zeros and more accurate numbers
- Default method: Interior point (`interior-point`)
  - faster for some inputs
- Look into `highs-*` for more highly optimized versions

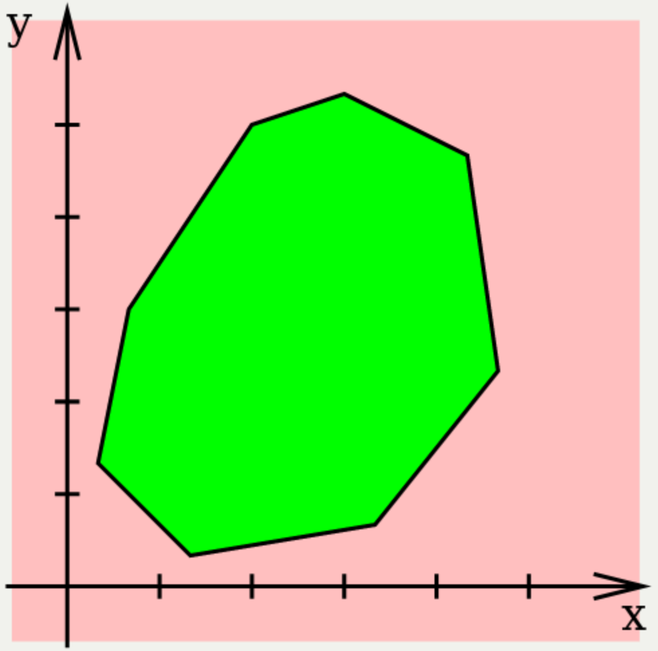
Digression: How to check if something is zero?

With finite–precision computer arithmetic, you should check if $|x| < 10^{-6}$ or another sufficiently small number

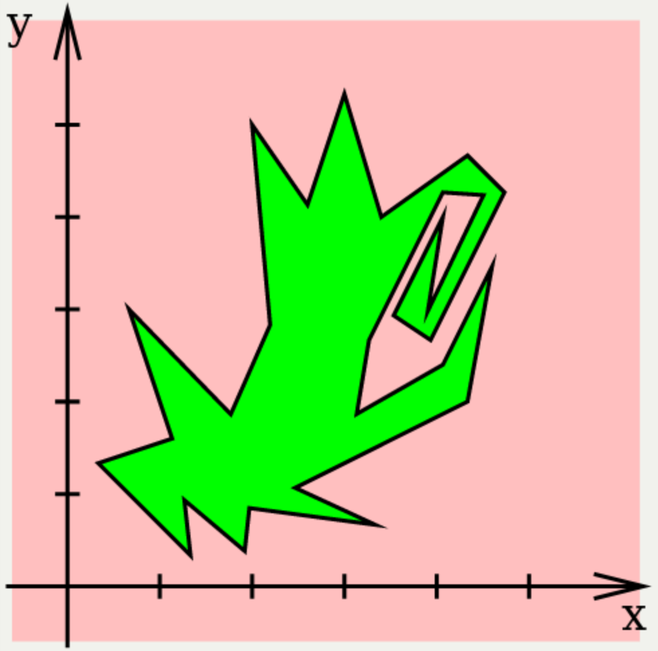
In [8]:
```
essentially_zero = 0.3 - 3 * 0.1
print("Test 1:", essentially_zero == 0)

# better test
def is_zero(x):
    return x < 1e-6 and x > -1e-6

print("Test 2:", is_zero(essentially_zero))
```

```
Test 1: False
Test 2: True
```

Convex vs. non-convex optimization



Convex



Non-convex

- Linear programming is an example of convex optimization
- Convex optimization often easier