2018202010

March 9, 2020

1 Assignment-1 Linear Programming

The objective of this assignment is to show the applications of linear programming in real life problems. You will be asked to solve problems from classical physics to puzzles.

1.1 Instructions

- For each question you need to write the formulation in markdown and solve the problem using `cvxpy`.
- Ensure that this notebook runs without errors when the cells are run in sequence.
- Plagarism will not be tolerated.
- Use only `python3` to run your code.
- If you are facing issues running the notebook on your local system. Use google collab to run the notebook online. To run the notebook online, go to google collab. Go to `File -> Upload Notebook` and import the notebook file

1.2 Submission

- Rename the notebook to <roll_number>.ipynb and submit **ONLY** the notebook file on moodle.

1.3 Problems

1. Sudoku
2. Best Polyhedron
3. Largest Ball
4. Illumination Problem
5. Jigsaw Puzzle

[1]:
```
# Installation dependencies
!pip3 install numpy==1.18.1 matplotlib==3.1.3 scipy==1.4.1 sklearn
!pip3 install cvxpy==1.0.25 scikit-image==0.16.2
```

```
Requirement already satisfied: numpy==1.18.1 in /usr/local/lib/python3.6/dist-
packages (1.18.1)
Requirement already satisfied: matplotlib==3.1.3 in
/usr/local/lib/python3.6/dist-packages (3.1.3)
Requirement already satisfied: scipy==1.4.1 in /usr/local/lib/python3.6/dist-
packages (1.4.1)
```

```
Requirement already satisfied: sklearn in /usr/local/lib/python3.6/dist-packages
(0.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.6/dist-
packages (from matplotlib==3.1.3) (0.10.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib==3.1.3) (1.1.0)
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib==3.1.3) (2.6.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib==3.1.3) (2.4.6)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.6/dist-
packages (from sklearn) (0.22.1)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages
(from cycler>=0.10->matplotlib==3.1.3) (1.12.0)
Requirement already satisfied: setuptools in /usr/local/lib/python3.6/dist-
packages (from kiwisolver>=1.0.1->matplotlib==3.1.3) (45.2.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-
packages (from scikit-learn->sklearn) (0.14.1)
Requirement already satisfied: cvxpy==1.0.25 in /usr/local/lib/python3.6/dist-
packages (1.0.25)
Requirement already satisfied: scikit-image==0.16.2 in
/usr/local/lib/python3.6/dist-packages (0.16.2)
Requirement already satisfied: multiprocess in /usr/local/lib/python3.6/dist-
packages (from cvxpy==1.0.25) (0.70.9)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.6/dist-
packages (from cvxpy==1.0.25) (1.4.1)
Requirement already satisfied: osqp>=0.4.1 in /usr/local/lib/python3.6/dist-
packages (from cvxpy==1.0.25) (0.6.1)
Requirement already satisfied: scs>=1.1.3 in /usr/local/lib/python3.6/dist-
packages (from cvxpy==1.0.25) (2.1.1.post2)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages
(from cvxpy==1.0.25) (1.12.0)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.6/dist-
packages (from cvxpy==1.0.25) (1.18.1)
Requirement already satisfied: ecos>=2 in /usr/local/lib/python3.6/dist-packages
(from cvxpy==1.0.25) (2.0.7.post1)
Requirement already satisfied: matplotlib!=3.0.0,>=2.0.0 in
/usr/local/lib/python3.6/dist-packages (from scikit-image==0.16.2) (3.1.3)
Requirement already satisfied: pillow>=4.3.0 in /usr/local/lib/python3.6/dist-
packages (from scikit-image==0.16.2) (6.2.2)
Requirement already satisfied: PyWavelets>=0.4.0 in
/usr/local/lib/python3.6/dist-packages (from scikit-image==0.16.2) (1.1.1)
Requirement already satisfied: networkx>=2.0 in /usr/local/lib/python3.6/dist-
packages (from scikit-image==0.16.2) (2.4)
Requirement already satisfied: imageio>=2.3.0 in /usr/local/lib/python3.6/dist-
packages (from scikit-image==0.16.2) (2.4.1)
Requirement already satisfied: dill>=0.3.1 in /usr/local/lib/python3.6/dist-
packages (from multiprocess->cvxpy==1.0.25) (0.3.1.1)
```

```
Requirement already satisfied: future in /usr/local/lib/python3.6/dist-packages
(from osqp>=0.4.1->cvxpy==1.0.25) (0.16.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.6/dist-
packages (from matplotlib!=3.0.0,>=2.0.0->scikit-image==0.16.2) (0.10.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib!=3.0.0,>=2.0.0->scikit-
image==0.16.2) (2.4.6)
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib!=3.0.0,>=2.0.0->scikit-
image==0.16.2) (2.6.1)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.6/dist-packages (from matplotlib!=3.0.0,>=2.0.0->scikit-
image==0.16.2) (1.1.0)
Requirement already satisfied: decorator>=4.3.0 in
/usr/local/lib/python3.6/dist-packages (from networkx>=2.0->scikit-
image==0.16.2) (4.4.1)
Requirement already satisfied: setuptools in /usr/local/lib/python3.6/dist-
packages (from kiwisolver>=1.0.1->matplotlib!=3.0.0,>=2.0.0->scikit-
image==0.16.2) (45.2.0)
```

[0]:

```
# Compatibility imports
from __future__ import print_function, division

# Imports
import os
import sys
import random

import numpy as np
import cvxpy as cp

import matplotlib.pyplot as plt

# Modules specific to problems
from sklearn.datasets import make_circles # For problem 2 (Best Polyhedron)
from scipy.spatial import ConvexHull # For problem 3 (Largest Ball in Polyhedron)
from scipy.linalg import null_space # For problem 4 (Illumination)
import matplotlib.cbook as cbook # For problem 5 (Jigsaw)
from skimage.transform import resize # For problem 5 (Jigsaw)
% matplotlib inline
```

1.4 Question-1 Sudoku

- In this problem you will develop a mixed integer programming algorithm, based upon branch and bound, to solve Sudoku puzzles as described in class.
- In particular, you need to implement the class SudokuSolver

The function takes as input a Sudoku puzzle as a 9x9 "list of lists" of integers, i.e.,

```
puzzle = [[4, 8, 0, 3, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 7, 1],
          [0, 2, 0, 0, 0, 0, 0, 0, 0],
          [7, 0, 5, 0, 0, 0, 0, 6, 0],
          [0, 0, 0, 2, 0, 0, 8, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 1, 0, 7, 6, 0, 0, 0],
          [3, 0, 0, 0, 0, 0, 4, 0, 0],
          [0, 0, 0, 0, 5, 0, 0, 0, 0]]
```

where zeros represent missing entries that must be assigned by your algorithm, and all other integers represent a known assignment.

- The class SudokuSolver inherits the Sudoku class. You need to make changes **only** to the SudokuSolver class. Write function `plot` to plot the unsolved and solved puzzle. Write function `solve` to create our own solver, the function can get the unsolved puzzle as the input as should return a 9x9 numpy array (solved puzzle), where solved puzzle contains the input puzzle with all the zeros assigned to their correct values. For instance, for the above puzzle this would be

  solved_puzzle = [[4, 8, 7, 3, 1, 2, 6, 9, 5], [5, 9, 3, 6, 8, 4, 2, 7, 1], [1, 2, 6, 5, 9, 7, 3, 8, 4], [7, 3, 5, 8, 4, 9, 1, 6, 2], [9, 1, 4, 2, 6, 5, 8, 3, 7], [2, 6, 8, 7, 3, 1, 5, 4, 9], [8, 5, 1, 4, 7, 6, 9, 2, 3], [3, 7, 9, 1, 2, 8, 4, 5, 6], [6, 4, 2, 9, 5, 3, 7, 1, 8]]

- You should write code to solve this problem using cvxpy.

Write the code in SudokuSolver class only.

[0]:
```
# Class Sudoku will generate new sudoku problems for you to solve. You cannot␣
↪change this code. Complete the formulation and the solver below
class Sudoku():
  def __init__(self):
    super(Sudoku,self).__init__()
    self.puzzle = None # Unsolved sudoku
    self.solution = None # Store the solution here
    pass

  def construct_solution(self):
    """
      This function created a 9x9 solved sudoku example.
      It can be used as a reference to see the performance of your solver.
    """
    while True: # until a solved sudoku puzzle if created
      puzzle = np.zeros((9,9))
      rows    = [set(range(1,10)) for i in range(9)] #   set of available
      columns = [set(range(1,10)) for i in range(9)] #   numbers for each
      squares = [set(range(1,10)) for i in range(9)] #   row, column and square

      try:
        for i in range(9): # for each roe
```

```
            for j in range(9): # for each column

                # Randomly choose a possible number for the location
                choices = rows[i].intersection(columns[j]).intersection(squares[(i//
↪3)*3 + j//3])
                choice  = random.choice(list(choices))

                puzzle[i,j] = choice        # update the puzzle

                # Remove from the choice from row,column, square
                rows[i].discard(choice)
                columns[j].discard(choice)
                squares[(i//3)*3 + j//3].discard(choice)

          # success! every cell is filled.
          return puzzle

        except IndexError:
          # if there is an IndexError, we have worked ourselves in a corner (we␣
↪just start over)
          continue

  def construct_problem(self,solution,n=28):
    """
      Construct the puzzle by removing a cell if it is possible to deduce a␣
↪cell's value from the remaining cells
      @param: n => minimum number of unplucked/remaining cells
    """

    def canBeDeduced(puz, i, j, c): # check if the cell can be deduced from the␣
↪remaining cells
      v = puz[c//9,c%9]
      if puz[i,j] == v: return True
      if puz[i,j] in range(1,10): return False

      for m in range(9): # test row, col, square
        # if not the cell itself, and the mth cell of the group contains the␣
↪value v, then "no"
        if not (m==c//9 and j==c%9) and puz[m,j] == v: return False
        if not (i==c//9 and m==c%9) and puz[i,m] == v: return False
        if not ((i//3)*3 + m//3==c//9 and (j//3)*3 + m%3==c%9) and puz[(i//3)*3␣
↪+ m//3,(j//3)*3 + m%3] == v:
          return False

      return True
```

```
    cells = set(range(81))
    cellsLeft = set(range(81))

    while len(cells) > n and len(cellsLeft): # Cells in the problem > n and␣
↪cells left to be plucked > 0
      cell = random.choice(list(cellsLeft)) # choose a random cell
      cellsLeft.discard(cell)

      # record whether another cell in these groups could also take
      # on the value we are trying to pluck
      row = col = square = False

      for i in range(9): # For all numbers
        if i != cell/9: # can be deduced from the row
          if canBeDeduced(solution, i, cell%9, cell): row = True
        if i != cell%9: # can be deduced from the col
          if canBeDeduced(solution, cell//9, i, cell): col = True
        if not (((cell//9)//3)*3 + i//3 == cell//9 and ((cell//9)%3)*3 + i%3 ==␣
↪cell%9): # can be deduced from the square
          if canBeDeduced(solution, ((cell//9)//3)*3 + i//3, ((cell//9)%3)*3 +␣
↪i%3, cell): square = True

      if row and col and square:
        continue # could not pluck this cell, try again.
      else:
        # this is a pluckable cell!
        solution[cell//9][cell%9] = 0 # 0 denotes a blank cell
        cells.discard(cell) # remove from the set of visible cells (pluck it)
        # we don't need to reset "cellsleft" because if a cell was not pluckable
        # earlier, then it will still not be pluckable now (with less information
        # on the board).

    return solution
```

Write the formulation of your solution here

$$\max 0$$

subject to constraints-

$$\sum_{v=1}^{9} x_{vrc} = 1 \; \forall r \in [1,9],\; c \in [1,9]$$

$$\sum_{r=1}^{9} x_{vrc} = 1 \; \forall v \in [1,9],\; c \in [1,9]$$

$$\sum_{c=1}^{9} x_{vrc} = 1 \ \forall v \in [1,9],\ r \in [1,9]$$

$$\sum_{r=3p-2}^{3p} \sum_{c=3q-2}^{3q} x_{vrc} = 1 \ \forall v \in [1,9],\ p,q \in [1,3]$$

$$x_{vrc} = 1 \ if \ puzzle_{vrc} = 1$$

$x_{vrc} \in \{0,1\}$

[0]:
```
# Create your sudoku puzzle solver here
class SudokuSolver(Sudoku):
  def __init__(self):
    super(SudokuSolver,self).__init__()
    self.solution = self.construct_solution() # Store the solution here
    self.puzzle = self.construct_problem(self.solution.copy(),n=28) # Unsolved␣
↪sudoku
    self.sol = {}
  def plot(self):

    n = 9
    for i in range(n):
      val_matrix = self.sol[i]
      for r in range(n):
        for c in range(n):
          x = val_matrix[r][c].value
          if x == 1:
            self.puzzle[r][c] = i + 1
    print('Solution:')
    print(self.puzzle)
    print("Original Solution")
    print(self.solution)

  def findKnowns(self):
        knowns = []
        r, c = self.puzzle.shape
        for row in range(r):
            for col in range(c):
                if self.puzzle[row][col] != 0:
                    knowns.append( ((row,col), self.puzzle[row][col]) )
        return knowns

  def solve(self):
     """
       Write your code here.
       The function should return the solved sudoku puzzle
     """

    n = 9
```

```
knowns = self.findKnowns()

#variables
X = {}
for i in range(n):
    X[i] = cp.Variable((n,n), boolean = True)

#fixing the known values
#value in each row and col should be 1
constraints = []
c1_1 = []
c1_2 = []
for (index, val) in knowns:
    r, c = index[0], index[1]
    c1_1 += [X[val-1][r][c] <= 1]
    c1_2 +=  [X[val-1][r][c] >= 1]

# print("value in each cell should exactly be 1")
val_sum = []
for row in range(n):
    s = 0
    for col in range(n):
        for val in range(n):
            s += X[val][row][col]
        val_sum.append(s)
        s = 0

val_sum = np.array(val_sum)
c5 = np.all(x == 1 for x in val_sum)

# print('value should appear only once in entire column')
col_sum = []
for val in range(n):
  s = 0
  for col in range(n):
    for row in range(n):
      s += X[val][row][col]
    col_sum.append(s)
    s = 0

col_sum = np.array(col_sum)
c2 = [i == 1 for i in col_sum]

# print('value should appear only once in entire row')
row_sum = []
for val in range(n):
    s = 0
```

```
        for row in range(n):
          for col in range(n):
              s += X[val][row][col]
          row_sum.append(s)
          s = 0

    row_sum = np.array(row_sum)
    c3 = [i == 1 for i in row_sum]

    # print('value should appear only once in entire block')
    block_sum = []
    for val in range(n):
      s = 0
      for r in range(0,n, 3):
        for c in range(0,n, 3):
          m = X[val]
          s = cp.sum(m[r:r+3, c:c+3]) # xvrc r , c 0 to 3
          block_sum.append(s)
          s = 0

    block_sum = np.array(block_sum)
    c4 = [i == 1 for i in block_sum]

    constraints += c5
    constraints += c1_1
    constraints += c1_2
    constraints += c2
    constraints += c3
    constraints += c4

    objective = cp.Maximize(0)

    problem = cp.Problem(objective, constraints)

    result = problem.solve(solver=cp.GLPK_MI)
    self.sol = X
    return
```

[5]:

```
solver = SudokuSolver()
solver.solve()
solver.plot()
```

```
Solution:
[[6. 5. 7. 9. 8. 4. 1. 3. 2.]
 [1. 4. 3. 2. 6. 5. 9. 7. 8.]
 [9. 2. 8. 3. 1. 7. 4. 6. 5.]
 [2. 3. 4. 7. 9. 1. 8. 5. 6.]
```

```
 [5. 8. 1. 4. 3. 6. 7. 2. 9.]
 [7. 9. 6. 8. 5. 2. 3. 1. 4.]
 [4. 7. 5. 1. 2. 9. 6. 8. 3.]
 [3. 1. 2. 6. 4. 8. 5. 9. 7.]
 [8. 6. 9. 5. 7. 3. 2. 4. 1.]]
Original Solution
[[6. 5. 7. 9. 8. 4. 1. 3. 2.]
 [1. 4. 3. 2. 6. 5. 9. 7. 8.]
 [9. 2. 8. 3. 1. 7. 4. 6. 5.]
 [2. 3. 4. 7. 9. 1. 8. 5. 6.]
 [5. 8. 1. 4. 3. 6. 7. 2. 9.]
 [7. 9. 6. 8. 5. 2. 3. 1. 4.]
 [4. 7. 5. 1. 2. 9. 6. 8. 3.]
 [3. 1. 2. 6. 4. 8. 5. 9. 7.]
 [8. 6. 9. 5. 7. 3. 2. 4. 1.]]
```

1.5 Question-2 Polyhedron

Explain how you would solve the following problem using linear programming. You are given two sets of points in Rn:

$$S1 = \{x_1, ..., x_N\}, S2 = \{y_1, ..., y_M\}.$$

You are asked to find a polyhedron

$$P = \{x | a_i^T x b_i, i = 1, ..., m\}$$

that contains the points in S1 in its interior, and does not contain any of the points in S2:

$$S1\{x | a_i^T x < b_i, i = 1, ..., m\}$$

$$S2\{x | a_i^T x > b_i foratleastonei\} = R_n - P.$$

An example is shown in the figure, with the points in S1 shown as open circles and the points in S2 as filled circles. You can assume that the two sets are separable in the way described.

- Your solution method should return a_i and b_i, i = 1, . . . , m, given the sets S1 and S2. The number of inequalities m is not specified, but it should not exceed 20, i.e your polyhedron should not have more than 20 faces.
- You are allowed to solve one or more LPs or LP feasibility problems. The method should be efficient, i.e., the dimensions of the LPs you solve should not be exponential as a function of N and M.
- You can calculate the quality of your solution by dividing the number of points in S1 your polyhedron is leaving out (points lying outside the polyhedron) by the total number of points in the set S1 (= N). The lower the value, the more efficient your solution will be. Use this metric to choose the most efficient solution out of all the possible solutions.

- The class PolyhedronSolver inherits the Polyhedron class. You need to make changes **only** to the PolyhedronSolver class. Write function `plot` to plot the points and the polyhedron (Look at question-3 on how to plot a polyhedron). Write function `solve` to create our own solver, the function can get the S1 & S2 as the input as should return a numpy array of size Dx2, where the D is the number the vertices of the polyhedron.

[0]:
```
class Polyhedron():
  def __init__(self):
    super(Polyhedron,self).__init__()
    data, labels = make_circles(n_samples=1000, noise=0.15,factor=0.3) # This␣
↪will create our data
    self.S1 = data[labels==0] # Points outside the polyhedron
    self.S2 = data[labels==1] # Points inside the polyhedron
```

Write the formulation of your solution here

$$\min r - \delta$$

subject to constraints-

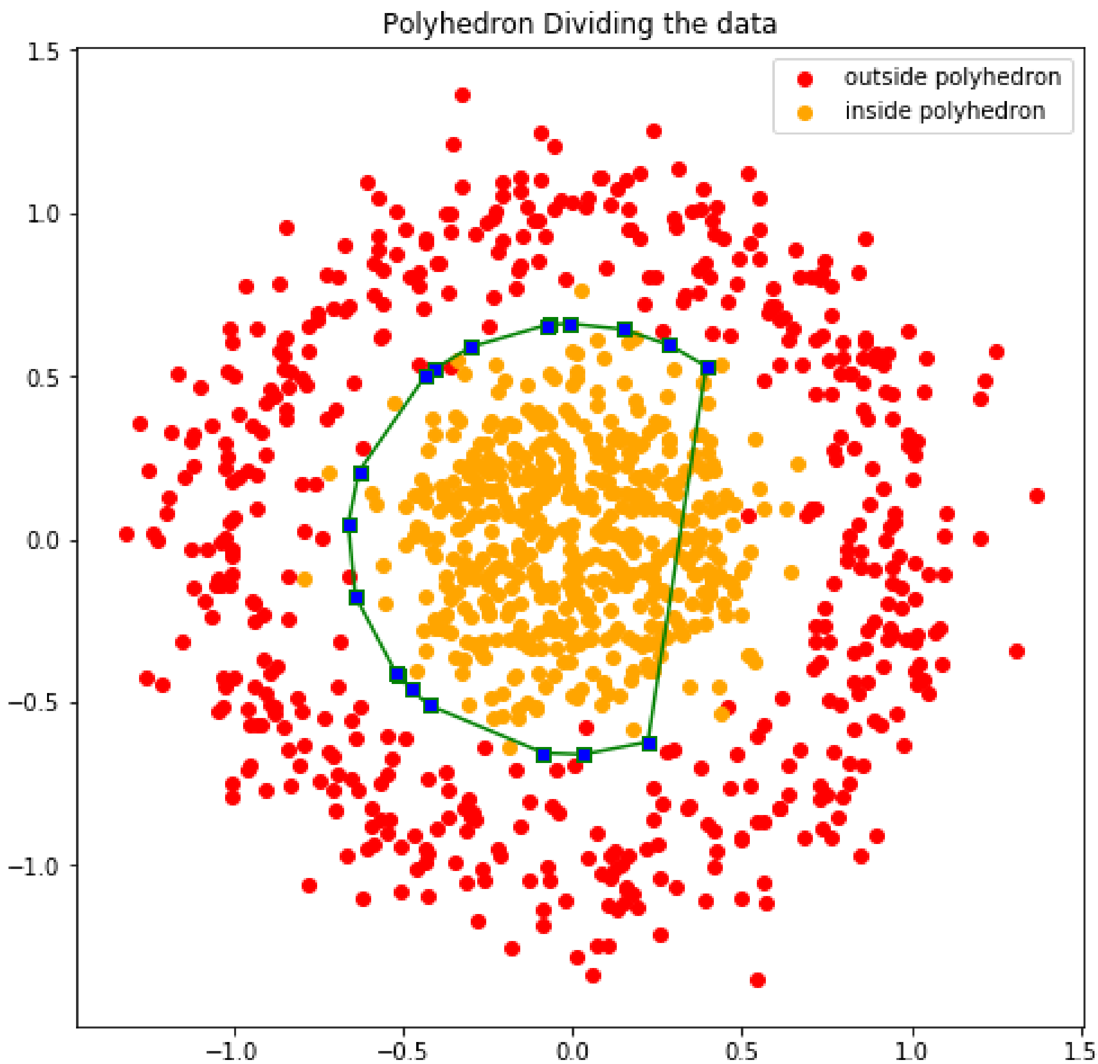
$$\|x_i\| \geq r + \delta \; \forall x_i \in S_1$$
$$\|x_i\| \leq r - \delta \; \forall x_i \in S_2$$
$$r \geq 0$$

[7]:
```
from numpy import linalg as LA
from scipy.spatial import ConvexHull
class PolyhedronSolver(Polyhedron):
  def __init__(self):
    super(PolyhedronSolver,self).__init__()
    self.R = cp.Variable()
    pass

  def plot(self):
    R = self.R.value
    fig = plt.figure(figsize=(8,8)) # Create 8x8 inches figure
    ax = fig.add_subplot(111) # Create a graph inside the figure
    ax.scatter(self.S1[:,0],self.S1[:,1],c="red",label="outside polyhedron") #␣
↪Plot S1
    ax.scatter(self.S2[:,0],self.S2[:,1],c="orange",label="inside polyhedron") #␣
↪PlotS2

    """
      Write code here for plotting your polyhedron
    """

    circle1 = plt.Circle((0, 0), R , fill=False)
```

```
#     ax.add_artist(circle1)

    #polar form
    theta = np.random.uniform(0,1,20) * 2 * np.pi

    random_points = []
    for t in theta:
      x = np.cos(t)*R
      y = np.sin(t)*R
      random_points.append([x,y])

    random_points = np.array(random_points)
    hull = ConvexHull(random_points)

    for simplex in hull.simplices:
      plt.plot(random_points[simplex, 0], random_points[simplex, 1],␣
↪color='green', linestyle='solid', marker='s', markerfacecolor='blue')
    ax.set_title("Polyhedron Dividing the data")
    ax.legend()
    plt.show()

  def solve(self):
    """
      Write your code here.

    """

    delta = cp.Variable()
    constraints = [self.R >= 0]

    #all pts outside the circle should have distance greater than equal to r
    c1 = []
    for point in self.S1: #outside red points
      dist = LA.norm(point)
      c1 += [dist >= self.R + delta]

    #all pts inside the circle should have distance less than equal to r
    c2 = []
    for point in self.S2: #inside orange points
      dist = LA.norm(point)
      c2 += [dist <= self.R - delta]

    constraints += c1
    constraints += c2

    objective = cp.Minimize(self.R - delta)
    problem = cp.Problem(objective, constraints)
```

```
    result = problem.solve(solver = cp.GLPK_MI)
    print("problem status {}, r value {}".format(problem.status, self.R.value))
    return
solver = PolyhedronSolver()
solver.solve()
solver.plot()
```

```
problem status optimal, r value 0.6613093422748014
```



1.6 Question-3 Largest Ball in a polyhedron

Find the largest ball

$$B(x_c, R) = \{x : ||xx_c||R\}$$

enclosed in a given polyhedron

$$P = \{x | a_i^T x b_i, i = 1, ..., m\}$$

- The problem variables are the center xc Rn and the radius R of the ball.
- The class CircleSolver inherits the CircleSolver class. You need to make changes only to the CircleSolver class. Write function `plot` to plot the polyhedron and the circle. Write function `solve` to create our own solver, the function can get the polyhedron as the input as should return a tuple (center,radius) where center is 1x2 numpy array containing the center of the circle, and radius is a scalar value containing the largest radius of the possible.

[0]:
```
class CircleInPolygon():
  def __init__(self):
     super(CircleInPolygon,self).__init__()
     self.polygon = np.random.random((10,2))
     self.polygon = self.polygon[ConvexHull(self.polygon).vertices,:] # A␣
 ↪polygon is stored here
```

Write the formulation of problem here

$$\max R$$

subject to constraints-

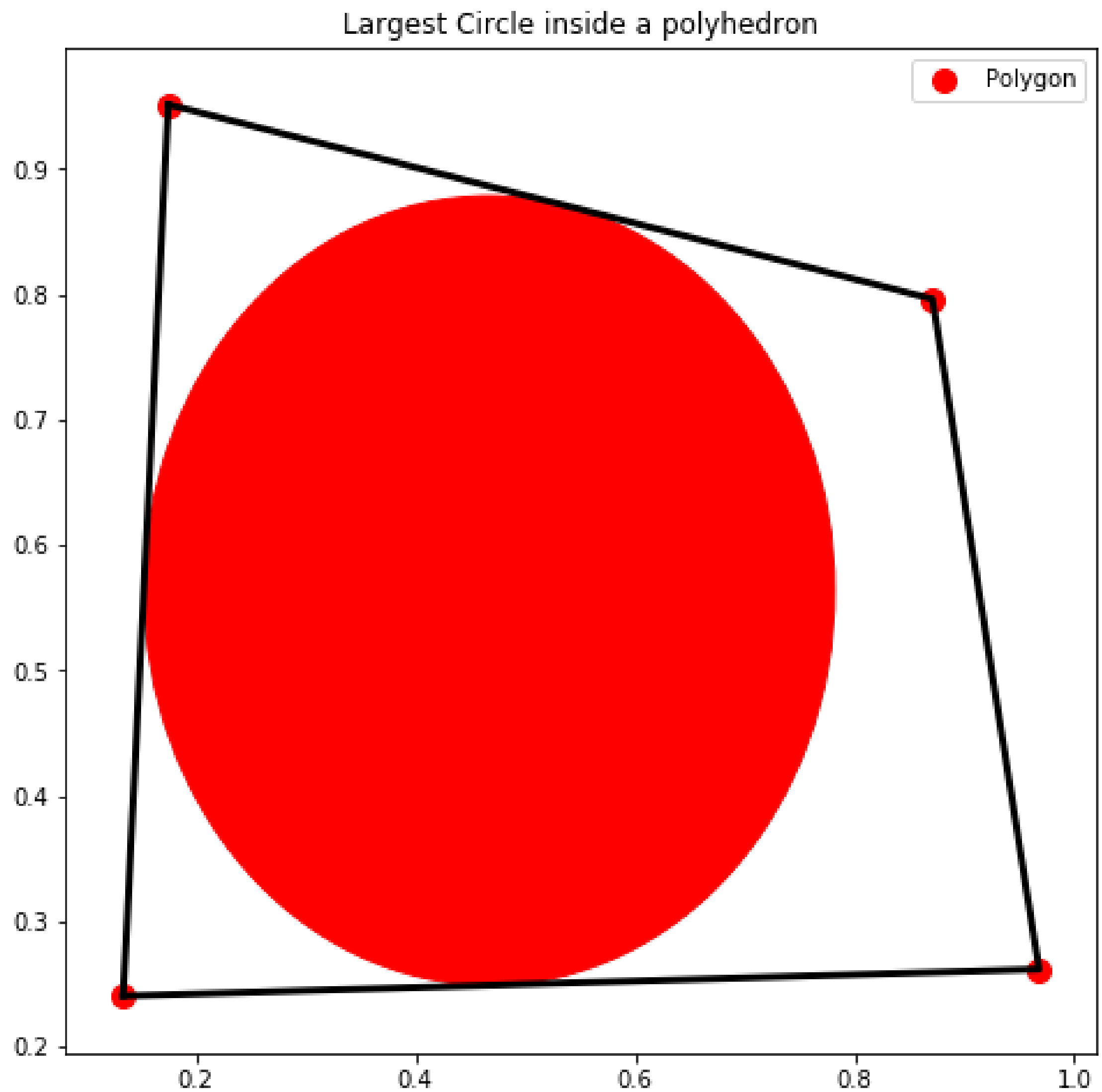
$$A_i * X_c + \|A_i\| * RB_i \; \forall i \in V$$

[9]:
```
# Create your circle puzzle solver here
import itertools

class CircleSolver(CircleInPolygon):
  def __init__(self):
      super(CircleSolver,self).__init__()
      self.X_c = cp.Variable(2)
      self.R = cp.Variable()

  def plot(self):

    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(111)
    ax.plot(self.polygon[:,0],self.polygon[:,1],linewidth=3,c="black") # Plot␣
 ↪the points
    ax.plot([self.polygon[0,0],self.polygon[-1,0]],[self.polygon[0,1],self.
 ↪polygon[-1,1]],linewidth=3,c="black") # Plot the edges
    ax.scatter(self.polygon[:,0],self.polygon[:
 ↪,1],s=100,c="red",label="Polygon") # Plot the edge connecting last and the␣
 ↪first point
```

```
    """
     Add code to plot the circle
    """
    circle1 = plt.Circle((self.X_c.value[0], self.X_c.value[1]), self.R.value,␣
↪color='r')
    ax.add_artist(circle1)

    ax.set_title("Largest Circle inside a polyhedron")
    plt.legend()
    plt.show()


  def plot_from_point(self,P0, P1):
    x1,y1 = P0[0],P0[1]
    x2,y2 = P1[0], P1[1]
    a = y2 - y1
    b = x1 - x2
    c = a*(x1) + b*(y1)
    return a,b,c

  def solve(self):

    constraints = []
    vertices = len(self.polygon)
    A = []
    B = []
    for i in range(vertices):
        # print('plotting b/w pt {} and pt {}'.format(self.polygon[i],self.
↪polygon[(i+1)%vertices]))
        a,b,c = self.plot_from_point(self.polygon[i%vertices],self.
↪polygon[(i+1)%vertices])
        A.append([a,b])
        B.append(c)


    A = np.array(A)
    B = np.array(B)

    for i in range(vertices):
      c = [A[i]*self.X_c + np.linalg.norm(A[i])*self.R <= B[i] ]
      constraints += c

    objective = cp.Maximize(self.R)
    problem = cp.Problem(objective, constraints)
    result = problem.solve()
    print(problem.status)
    print("X_c: {}, R: {}".format(self.X_c.value, self.R.value))
```

```
solver = CircleSolver()
solver.solve()
solver.plot()
```

```
optimal
X_c: [0.49281656 0.5441654 ], R: 0.20010129856565767
```



1.7 Question-4 Illumination Problem

We consider an illumination system of m lamps, at positions l1, . . . , lm R2, illuminating n flat patches. The patches are line segments; the ith patch is given by

$$[v_i, v_i + 1]$$

where v1, . . . , vn+1 R2. The variables in the problem are the lamp powers p1, . . . , pm, which can vary between 0 and 1. The illumination at (the midpoint of) patch i is denoted Ii. We will use a simple model for the illumination:

$$Ii = \sum_{j=1}^{m} a_{ij} p_j$$

$$a_{ij} = r_{ij}^2 (max(cos_{ij}, 0))$$

where rij denotes the distance between lamp j and the midpoint of patch i, and ij denotes the angle between the upward normal of patch i and the vector from the midpoint of patch i to lamp j.

This model takes into account "self-shading" (i.e., the fact that a patch is illuminated only by lamps in the halfspace it faces) but not shading of one patch caused by another. Of course we could use a more complex illumination model, including shading and even reflections. This just changes the matrix relating the lamp powers to the patch illumination levels.

The problem is to determine lamp powers that make the illumination levels close to a given desired illumination level Ides, subject to the power limits 0 pi 1. Suppose we use the maximum deviation

$$(p) = max_{k=1,\ldots,n} |I_k I_{des}|$$

as a measure for the deviation from the desired illumination level. Formulate the illumination problem using this criterion as a linear programming problem.

Create the data using the *Illumination* class and solve the problem using IlluminationSolver class. The elements of A are the coefficients aij in the above equation.

Compute a feasible p using this first method, and calculate (p)

[0]:
```
class Illumination():
  def __init__(self):
    super(Illumination,self).__init__()

    # Lamp position
    self.Lamps = np.array([[0.1 ,0.3, 0.4, 0.6 ,0.8 ,0.9 ,0.95],[1.0, 1.1, 0.6␣
↪,0.9, 0.9 ,1.2, 1.00]])
    self.m = self.Lamps.shape[1]     # number of lamps

    # begin and endpoints of patches
    self.patches = [np.arange(0,1,1/12),np.array([0 ,0.1 ,0.2, 0.2, 0.1, 0.2 ,0.
↪3 ,0.2 ,0 ,  0 ,  0.2, 0.1])]
    self.patches = np.array(self.patches)
    self.n = self.patches.shape[1] -1  # number of patches

    # desired illumination
```

```
    Ides = 2;

    # construct A
    self.dpatches = self.patches[:,1:] - self.patches[:,:-1];  # tangent to␣
↪patches
    self.patches_mid = self.patches[:,1:] - 0.5*self.dpatches;         #␣
↪midpoint of patches
    A = np.zeros((self.n,self.m));
    for i in range(self.n):
      for j in range(self.m):
        dVI = self.Lamps[:,j]-self.patches_mid[:,i] # Find the distance between␣
↪each lamp and patch
        rij = np.linalg.norm(dVI,ord=2) # Find the radius/distance between lamp␣
↪and the midpoint of the patch
        normal = null_space(self.dpatches[:,i].reshape(1,2)) # Find the normal

        if normal[1] < 0: # we want an upward pointing normal
          normal = -1*normal
        A[i,j] = dVI.dot(normal)/(np.linalg.norm(dVI,ord=2)*np.linalg.
↪norm(normal,ord=2))/(rij**2); # Find A[i,j] as defined above
        if A[i,j] < 0:
          A[i,j] = 0

    self.A = A
```

Write the formulation of problem here

$$\min t$$

subject to constraints-

$$0 \leq P_i \leq 1 \; \forall i \in 1,2..,m$$

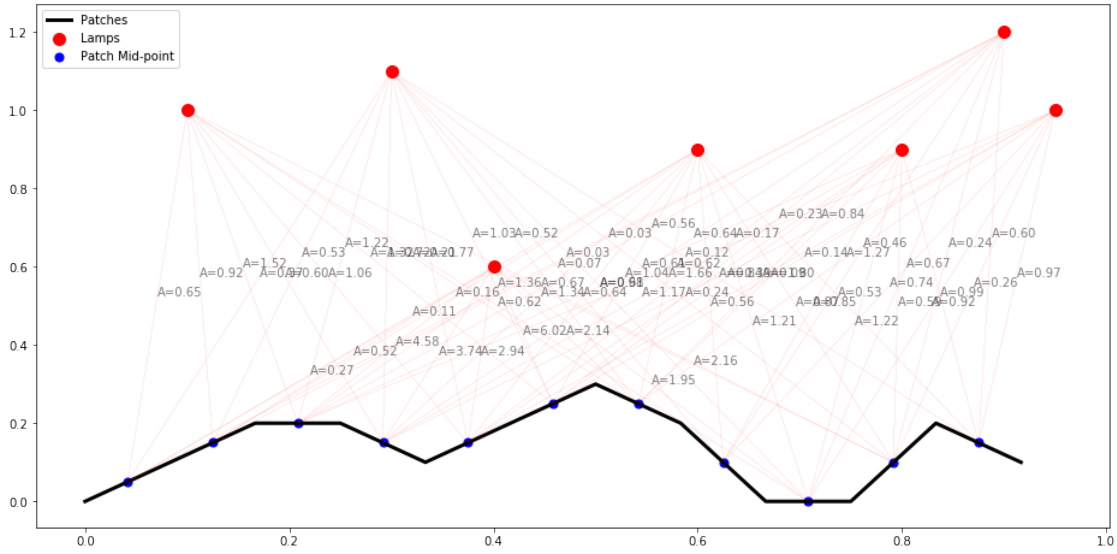
$$|\sum_{i=1}^{n} A_{ij} * P_j - Ides| \leq t \; \forall j \in 1,2..,m$$

[11]:
```
# Create your illumination solver here
class IlluminationSolver(Illumination):
  def __init__(self):
    super(IlluminationSolver,self).__init__()
    self.P = cp.Variable(self.m)
    self.t = cp.Variable()
  def plot(self):
    fig = plt.figure(figsize=(16,8))
    ax = fig.add_subplot(111)
```

```
    ax.scatter(self.Lamps[0,:],self.Lamps[1,:],s=100,c="red",label="Lamps") #␣
↪Lamps
    ax.scatter(self.patches_mid[0,:],self.patches_mid[1,:
↪],s=50,c="blue",label="Patch Mid-point") # Lamps
    ax.plot(self.patches[0,:],self.patches[1,:
↪],linewidth=3,c="black",label="Patches") # Patches

    # Normal joining lamps and patchs
    for i in range(self.n):
      for j in range(self.m):
        if self.A[i,j] > 0:
          ax.plot([self.Lamps[0,j], self.patches_mid[0,i]],[self.Lamps[1,j],␣
↪self.patches_mid[1,i]],'r--',linewidth=0.1,alpha=1)
          ax.text((self.Lamps[0,j]+self.patches_mid[0,i])/2,(self.Lamps[1,j] +␣
↪self.patches_mid[1,i])/2,"A={0:.2f}".format(self.A[i,j]),alpha=0.5)

    #calculate intensity of each lamp
    intensity = []
    for patches in range(self.n):
      s = 0
      for lamps in range(self.m):
        s += self.A[patches][lamps] * self.P.value[lamps]
      intensity.append(s)
    intensity = np.array(intensity)

    Ides = 2

    #compute difference of intensity and Ides
    differences = []
    for i in range(self.n):
      x = np.abs(intensity[i] - Ides)
      differences.append(x)
    differences = np.array(differences)

    #desired illumination
    print("phi (p) {}".format( np.max(differences)))

    plt.legend()
    plt.show()

  def solve(self):
    Ides = 2


    intensity = []
    for patches in range(self.n):
      s = 0
```

```
      for lamps in range(self.m):
        s += self.A[patches][lamps] * self.P[lamps] # I_lamp = Sum (A_patch,lamp␣
↪* P_lamp )
      intensity.append(s)

    intensity = np.array(intensity)

    differences = []
    for i in range(self.n):
      x = cp.abs(intensity[i] - Ides)
      differences.append(x) # I_lamp - Ides

    differences = np.array(differences)

    objective = cp.Minimize(self.t)
    constraints = [ self.P >= 0, self.P <= 1]
    for i in differences:
      c = [i <= self.t] # I_lamp - Ides <= t for all lamps
      constraints += c

    problem = cp.Problem(objective, constraints)
    result = problem.solve(solver=cp.GLPK_MI)
    print(problem.status)
    print("P values {} \nt value {}".format( self.P.value, self.t.value))

solver = IlluminationSolver()
solver.solve()
solver.plot()
```

```
optimal
P values [1.         0.66370061 0.         0.         0.         0.26521747
 1.        ]
t value 1.0910372833138409
phi (p) 1.0910372833138409
```


Patches
Lamps
Patch Mid-point
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0