Zoutendijk's Method for Constrained Optimization

14075065 – Manish Kumar Singh
14074017 – Shivam Garg
14074012 – Robin Khurana
14075061 – Janvijay Singh
14075010 – Atishay Jain

November 17, 2017

1 Introduction

Zoutendijk's Method is a method for iteratively finding the solution to a constrained optimization problem where the feasible region is defined by a system of linear or non-linear inequality constraints:

$$\begin{aligned} &\text{Minimize} \quad f(x) \\ &\text{subject to } g_i(x) \leq 0 \text{ for } i = 1, \ldots, m \end{aligned}$$

The functions $f(x)$ and $g_i(x)$ may be linear or non-linear in x. The method proceeds by finding an optimal improving feasible direction and then finds the optimal point for the mizimization problem in that direction starting from an initial point. The complete algorithm proceeds as described in Algorithm 1. The simplex method is used to find the optimal improving feasible direction, and then any line search technique can be used for finding the optimal point of the minimization problem in the optimal improving feasible direction so found.

Algorithm 1 Zoutendijk's algorithm

procedure OPTIMIZE
Initialization Step:
    Choose a starting point x_1 such that $g_i(x_1) \leq 0$ for $i = 1, \ldots, m$;
    $x_1 \leftarrow 1$;
    goto *Main Steps*.
Main Steps:
Step 1:Determine a feasible direction d_k
    Solve the following problem, where $I = \{i \mid g_i(x_k) = 0\}$:

$$\begin{array}{lll} \text{Minimize} & z & \\ \text{subject to} & \nabla f(x_k)^\top d - z \leq 0 & \\ & \nabla g_i(x_k)^\top d - z \leq 0 & \text{for } i \in I \\ & -1 \leq d_j \leq 1 & \text{for } j = 1, \ldots n. \end{array}$$

    Let (z_k, d_k) be an optimal solution.
    if $z_k = 0$ **then**
        stop.
    if $z_k < 0$ **then**
        goto *Step 2*.
Step 2:
    Let λ_k be an optimal solution to the following line search problem:

$$\begin{array}{ll} \text{Minimize} & f(x_k + \lambda_k d_k) \\ \text{subject to} & 0 \leq \lambda \leq \lambda_{max} \\ \text{where } \lambda_{max} = \sup\{\lambda : g_i(x_k + \lambda_k d_k) \leq 0 & \text{for } i = 1, \ldots, m\} \end{array}$$

    $x_{k+1} \leftarrow x_k + \lambda_k d_k$.
    $k \leftarrow k + 1$.
    goto *Step 1*.

2 Implementation

Finding the optimal improving feasible direction: The constrained optimization problem for finding the optimal feasible direction is given by:

$$\begin{array}{lll} \text{Minimize} & z & \\ \text{s.t.} & \nabla f(x_k)^\top d - z \leq 0 & \\ & \nabla g_i(x_k)^\top d - z \leq 0 & \text{for } i \in I \\ & -1 \leq d_j \leq 1 & \text{for } j = 1, \ldots n. \end{array}$$

which can be restated as the following linear programming problem, which can be solved using the simplex method as given in [2]:

$$\begin{array}{ll} \text{Minimize} & y^\top b \\ \text{s.t.} & y^\top A \geq c^\top \\ & y \geq 0 \end{array}$$

with

$$\begin{aligned} y^\top &= \begin{pmatrix} u_1 & \cdots & u_n & v_1 & \cdots & v_n & z_u & z_v \end{pmatrix}_{[1\times(2n+2)]}, \\ b^\top &= \begin{pmatrix} 0 & \cdots & 0 & 0 & \cdots & 0 & 1 & -1 \end{pmatrix}_{[1\times(2n+2)]}, \\ c^\top &= \begin{pmatrix} 0 & \underbrace{0 \quad \cdots \quad 0}_{|I| \text{ times}} & \underbrace{-1 \quad \cdots \quad -1}_{2n \text{ times}} \end{pmatrix}_{[1\times(2n+|I|+1)]}, \end{aligned}$$

where $d_j = u_j - v_j$ for $j = 1, \ldots, n$ and $z = z_u - z_v$, and

$$A = \begin{pmatrix} -\nabla f(x) & -\nabla g_{i_1}(x) & \cdots & -\nabla g_{i_{|I|}}(x) & +I_n & -I_n \\ +\nabla f(x) & +\nabla g_{i_1}(x) & \cdots & +\nabla g_{i_{|I|}}(x) & -I_n & +I_n \\ +1 & +1 & \cdots & +1 & 0_{1\times n} & 0_{1\times n} \\ -1 & -1 & \cdots & -1 & 0_{1\times n} & 0_{1\times n} \end{pmatrix}_{[(2n+2)\times(2n+|I|+1)]}$$

where I_n is an n dimensional identity matrix, $0_{1\times n}$ is a zero matrix of order $1 \times n$ and $|I|$ is the size of the active set I.

Finding the optimal point in the direction d: The golden search technique was used to find the optimal point in the optimal improving feasible direction, d.

3 Test Problems

We test our implementation on two test problems.

3.1 Problem – 1

The following problem was taken from [1]. The performance of the Zoutendijk's method on this problem is shown in Fig. 1.

$$\begin{aligned} &\text{Minimize} && 2x_1^2 + 2x_2^2 - 2x_1x_2 - 4x_1 - 6x_2 \\ &\text{s.t.} && x_1 + 5x_2 - 5 \leq 0 \\ & && 2x_1^2 - x_2 \leq 0 \\ & && -x1 \leq 0 \\ & && -x2 \leq 0 \end{aligned} \tag{1}$$

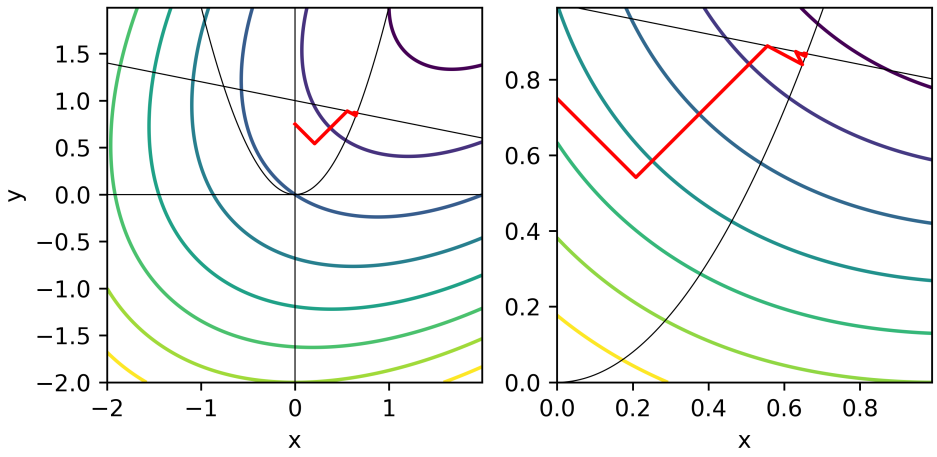

Figure 1: This figure shows the convergence of the algorithm on Problem 1. It starts from $(0, 0.75)$ and converges to the optimal point $(0.65850.86827)$ in 11 iterations. The zig-zag line shows how the algorithm moves. The thin lines show the constraints and the thick lines show the contour plot of the function to minimize.

3.2 Problem – 2

The following function is the Rosenbrock function constrained to a disk. The performance of the Zoutendijk's method on this problem is shown in Fig. 2.

$$\begin{aligned} &\text{Minimize} && (1-x)^2 + 100(y-x^2)^2 \\ &\text{s.t.} && x^2 + y^2 - 2 \leq 0 \leq 0 \end{aligned} \tag{2}$$

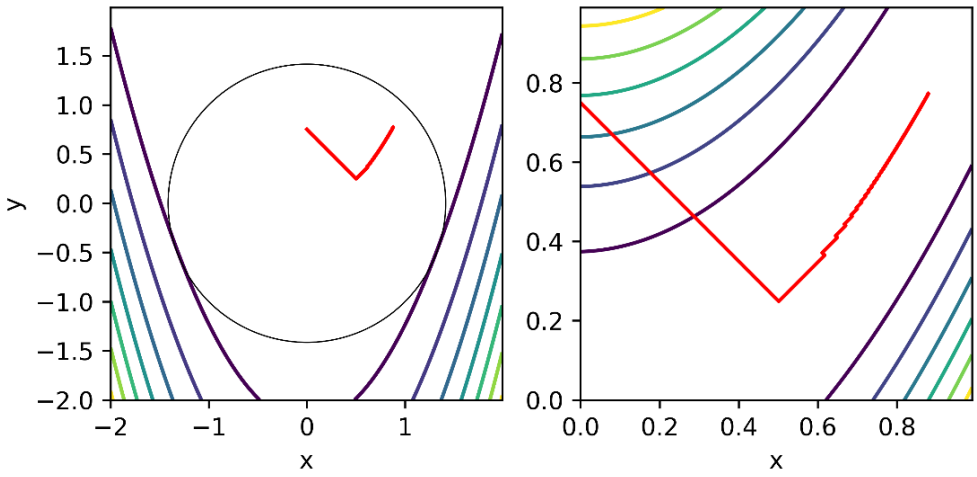

Figure 2: This figure shows the convergence of the algorithm on Problem 2 (Rosenbrock function constrained to a disk). It starts from $(0, 0.75)$ and converges to the optimal point $(0.8792, 0.7726)$ in 136 iterations. The zig-zag line shows how the algorithm moves. The thin lines show the constraints and the thick lines show the contour plot of the function to minimize.

4 Code

simplex.py

```
import numpy as np
import math

def solve(M, n):
    i = 0
    PIV = []
    while True:
        flag = True
        for ii in range(i, M.shape[0]-1):
            if M[ii, -1] < 0:
```

```
            flag = False
            break
        else:
            i = ii
    if flag:
        break # all b_i s are positive

    # b_ii is the first negative b_i == b_k
    FEASIBLE = False
    for jj in range(0, M.shape[1]-1):
        if M[ii, jj] < 0:
            FEASIBLE = True
            break

    if not FEASIBLE:
        return None, None, None

    # pivot is in column jj
    piv_col = jj
    piv_row = ii
    MIN = M[ii, -1]/M[ii, piv_col]
    for iii in range(M.shape[0]-1):
        if M[iii, -1] >= 0 and M[iii, piv_col] > 0:
            tmp = M[iii, -1]/M[iii, piv_col]
            if tmp < MIN:
                piv_row = iii
                tmp = MIN

    #print(piv_row, piv_col)
    # take pivot about piv_row, piv_col
    PIV.append((piv_row, piv_col))
    M = pivot(M, piv_row, piv_col)
    #print(M, piv_row, piv_col)

# all b_i s are positive
#print('bi done')

j = 0
while True:
    flag = True
    for jj in range(j, M.shape[1]-1):
```

```
            if M[-1, jj] < 0:
                flag = False
                break
            else:
                j = jj
        if flag:
            break # all c_i s are positive

        piv_col = jj
        piv_row = -1
        MIN =1e8
        FEASIBLE = False
        for iii in range(0, M.shape[0]-1):
            if M[iii, piv_col] > 0:
                FEASIBLE = True
                tmp = M[iii, -1]/M[iii, piv_col]
                if tmp < MIN:
                    piv_row = iii
                    MIN = tmp

        if not FEASIBLE:
            return None, None, None

        # take pivot about piv_row, piv_col
        PIV.append((piv_row, piv_col))
        M = pivot(M, piv_row, piv_col)
        #print(M, piv_row, piv_col)

    for (i, j) in PIV[::-1]:
        tmp = M[i, -1]
        M[i, -1] = M[-1, j]
        M[-1, j] = tmp

    X = np.copy(M[-1, :])[0:-1]
    Y = np.copy(M[:, -1])[0:-1]

    V = M[-1, -1]

    return X,Y,V

def pivot(M, piv_row, piv_col):
```

```
    X = np.zeros(M.shape)
    piv = M[piv_row, piv_col]

    for i in range(M.shape[0]):
        for j in range(M.shape[1]):
            if i == piv_row and j == piv_col:
                X[i, j] = 1/piv

            elif i == piv_row:
                X[i, j] = M[i, j]/piv

            elif j == piv_col:
                X[i, j] = -1*M[i, j]/piv

            else:
                X[i, j] = M[i, j] - M[piv_row, j]*M[i, piv_col]/piv

    return X
```

GoldenSectionMethod.py

```
from __future__ import division
from math import sqrt

phi = (1 + sqrt(5))/2

def goldenSectionSearch(function, xLow, xHigh, allowedError) :
''' Golden Search method to find minimum of funtion in range [xLow, xHigh]
-> funtion x is assumed to be unimodal in range [xLow, xHigh] '''

x1 = xHigh - (xHigh - xLow) / phi
x2 = xLow  + (xHigh - xLow) / phi

while abs(x2-x1) > allowedError :
if function(x1) < function(x2) :
xHigh = x2
else :
xLow  = x1

x1 = xHigh - (xHigh - xLow) / phi
x2 = xLow  + (xHigh - xLow) / phi
```

```
return (xLow + xHigh) / 2
```

main.py

```
import numpy as np
import simplex
from GoldenSectionMethod import *

def f_func(X):
x1=X[0]
x2=X[1]
return 2*(x1**2)+2*(x2**2)-2*x1*x2-4*x1-6*x2

def g_list(X):
x1=X[0]
x2=X[1]
return np.array([x1+5*x2-5, 2*x1**2 - x2, -x1, -x2])

def find_active_set(X):
print "Input to g_list",X
return [i for (i,c) in enumerate(X) if abs(c)<1e-3 ]

def grad_f(X):
diff=np.zeros((X.size,X.size))
for i in xrange(X.size):
diff[i][i]=1e-6
return np.resize(np.array([(f_func(X+diff[i])-f_func(X-diff[i]))/2e-6
                        for i in xrange(X.size)]), (X.size,1))

def grad_g(X):
diff=np.zeros((X.size,X.size))
for i in xrange(X.size):
diff[i][i]=1e-6
n=g_list(X).size
return np.array([[(g_list(X+diff[i])[j]-g_list(X-diff[i])[j])/2e-6
                         for i in xrange(X.size)] for j in xrange(n)])


X_list=np.zeros((1,2))
X0=np.array((0.000000, 0.000000))
X_list[0]=X0
```

```
z=10
i=0
n=X_list[0].size
while True:
current=X_list[i]
print "current:", current
active_set=find_active_set(g_list(current))
print "active_set:", active_set
f_grad=grad_f(current)
g_grad=grad_g(current)
a11=-1*f_grad
g_i=np.array([g_grad[a] for a in active_set])
g_i=g_i.T
a12=-1*g_i
if(len(active_set)>0):
A1=np.concatenate((a11, a12), axis=1)
else:
A1=a11
a21=f_grad
a22=g_i
if(len(active_set)>0):
A2=np.concatenate((a21, a22), axis=1)
else:
A2=a21
A1=np.concatenate((A1, np.eye(n), -1*np.eye(n)), axis=1)
A2=np.concatenate((A2, -1*np.eye(n), np.eye(n)), axis=1)
A3=np.ones((1, (1+len(active_set))))
A3=np.concatenate((A3, np.zeros((1, 2*n))), axis=1)
A4=-1*A3
A=np.concatenate((A1, A2, A3, A4), axis=0)
c=np.concatenate((np.zeros((1,1+len(active_set))),
                          np.ones((1,2*n))), axis=1)
final_matrix_1=np.concatenate((A,c), axis=0)
final_matrix_2=np.zeros((2*n+3, 1))
final_matrix_2[-2]=-1
final_matrix_2[-3] = 1
A=np.concatenate((final_matrix_1, final_matrix_2), axis=1)
print "A:", A
x,y,z = simplex.solve(A, n)
d=y[0:n]-y[n:2*n]
print "d:", d
```

```
if (abs(z)<=1e-8):
break
lambda_max =1
while(lambda_max>=0):
temp=g_list(current+lambda_max*d)
if (np.sum(temp>0)==0):
break
lambda_max -= 1e-6
print "lambda_max:", lambda_max
lambda_ans=goldenSectionSearch(lambda x : f_func(X_list[i]+x*d),
                                                    0,lambda_max , 1e-5)
print "lamda_ans:", lambda_ans
X_list=np.concatenate((X_list, np.resize(current+lambda_ans*d,
                                                  (1, n))), axis=0)
print "X_list["+str(i)+"]", X_list[i]
print "Iteration Complete"
i+=1

print X_list
```

5 References

1. **Nonlinear programming: theory and algorithms**, Bazaraa, Mokhtar S and Sherali, Hanif D and Shetty, Chitharanjan M (2013), John Wiley & Sons

2. **Linear Programming: A concise Introduction**, Thomas S. Ferguson, `https://www.math.ucla.edu/~tom/LP.pdf`