1. 二分法

## 代码

# T1: dichotomy

def bisection(f, a, b, epsilon = 1e-4, max\_iter = 100):

if a >= b:

raise ValueError("b must be bigger than a")

if f(a) \* f(b) >= 0:

raise ValueError("f(a) \* f(b) must be negative")

fa, fb = f(a), f(b)

for iter\_cnt in range(max\_iter):

x = (a + b) / 2

fx = f(x)

error = np.abs(fx)

if error < epsilon:

return x, error, iter\_cnt

if fa \* fx < 0:

b = x

fb = fx

else:

a = x

fa = fx

return x, error, iter\_cnt

def f(x):

return x\*\*4/4 - 4\*x\*\*3/3 + 5\*x\*\*2/2 - 2\*x

def df(x):

return x\*\*3 - 4\*x\*\*2 + 5\*x - 2

root, \_, \_ = bisection(df, 0, 4)

print(f"{root}")

## 结果

2.0

验证：，确是极小点

1. 0.618法

## 代码

# T2: 0.618

def f(x):

return np.exp(-x) + x\*\*2

def goldencut(f, a, b, epsilon = 1e-4, max\_iter = 100):

if a >= b:

raise ValueError("b must be bigger than a")

t = (np.sqrt(5) - 1)/2

for iter\_cnt in range(max\_iter):

error = np.abs(b - a)

if error < epsilon:

return (a + b)/2, error, iter\_cnt

x1 = a + (b - a)\*(1 - t)

x2 = a + (b - a)\*t

if f(x1) < f(x2):

b = x2

else:

a = x1

return (a + b)/2, error, iter\_cnt

root, \_, \_ = goldencut(f, -2, 3)

print(f"{root}")

## 结果

0.35173474094718815

验证：，确是极小点

1. Newton法

## 代码

# T3: Newton

def f(x):

return x\*\*4 - 4\*x\*\*3 - 6\*x\*\*2 - 16\*x + 4

def df(x):

return 4\*x\*\*3 - 12\*x\*\*2 - 12\*x - 16

def ddf(x):

return 12\*x\*\*2 - 24\*x - 12

def newton(f, x, df, epsilon = 1e-4, max\_iter = 100):

for iter\_cnt in range(max\_iter):

fx = f(x)

error = np.abs(fx)

if error < epsilon:

return x, error, iter\_cnt

x = x - fx/df(x)

return x, error, iter\_cnt

root, \_, \_ = newton(df, 2.5, ddf)

print(f"{root}")

root, \_, \_ = newton(df, 3, ddf)

print(f"{root}")

## 结果

4.000000000017841

4.000000184090883

验证：，确是极小点

1. 无约束优化

## 代码

# T4: steepest descent, conjugate gradient, DFP

class OptimizationAlgorithm:

pass

def StepSelect\_Newton(x, grad\_f, Hesse\_f, p, step, epsilon = 1e-7, max\_iter = 5):

for \_ in range(max\_iter):

fstep = np.dot(grad\_f(x + step\*p), p)

error = np.abs(fstep)

if error < epsilon:

break

# dlambda f(x + lambda\*p) = grad\_f(x + lambda\*p)\*p

step = step - fstep/np.dot(Hesse\_f(x + step\*p)@p, p)

return step, x + step\*p

def StepSelect\_GoldenCut(f, a, b, epsilon = 1e-7, max\_iter = 100):

if a >= b:

raise ValueError("b must be bigger than a")

t = (np.sqrt(5) - 1)/2

for \_ in range(max\_iter):

error = b - a

if error < epsilon:

break

x1 = a + (b - a)\*(1 - t)

x2 = a + (b - a)\*t

if f(x1) < f(x2):

b = x2

else:

a = x1

return (a + b)/2

# Steepest Descent

def SteepestDescent(f, grad\_f, x, StepSelectMethod, Hesse\_f, init\_step = 0.1, epsilon = 1e-7, max\_iter = 100):

iter\_cnt = 0

for \_ in range(max\_iter):

g = grad\_f(x)

p = -g

if np.linalg.norm(g) < epsilon:

break

# find step = argmin\_step f(x + step\*p)

if StepSelectMethod == "fixed":

step = init\_step

elif StepSelectMethod == "Newton":

step, \_ = StepSelect\_Newton(x, grad\_f, Hesse\_f, p, 0)

x = x + step\*p

iter\_cnt += 1

return x, f(x), iter\_cnt

# Conjugate Gradient (FR)

class ConjugateGradient\_FR():

def \_\_init\_\_(self, x, f, epsilon = 1e-7, max\_iter = 100):

self.x = x

self.f = f

self.epsilon = epsilon

self.max\_iter = max\_iter

self.step\_select\_methods = {

"fixed": {"init\_step": 0.1},

"Newton": {"Hesse\_f": None},

}

def optimize(self, grad\_f, StepSelectMethod, \*\*kwargs):

if StepSelectMethod not in self.step\_select\_methods:

raise ValueError(f"Unknown step select method: {StepSelectMethod}")

x = self.x

f = self.f

epsilon = self.epsilon

max\_iter = self.max\_iter

n = np.size(x)

g = grad\_f(x)

iter\_cnt = 0

if np.linalg.norm(g) >= epsilon:

if StepSelectMethod == "fixed":

pass

elif StepSelectMethod == "Newton":

Hesse\_f = kwargs["Hesse\_f"]

end = False

for \_ in range(max\_iter):

p = -g

iter\_in = 0

while(1):

# find step = argmin\_step f(x + step\*p)

if StepSelectMethod == "fixed":

step = self.step\_select\_methods["fixed"]["init\_step"]

x = x + step\*p

elif StepSelectMethod == "Newton":

# dlambda f(x + lambda\*p) = dot(grad\_f(x + lambda\*p), p)

# ddlambda f(x + lambda\*p) = dot(Hesse\_f(x + lambda\*p)@p, p)

\_, x = StepSelect\_Newton(x, grad\_f, Hesse\_f, p, 0)

g\_pre = g

g = grad\_f(x)

if np.linalg.norm(g) < epsilon:

end = True

break

iter\_in += 1

if iter\_in == n:

break

p = -g + (np.linalg.norm(g)/np.linalg.norm(g\_pre))\*\*2\*p

if end == True:

break

iter\_cnt += 1

return x, f(x), iter\_cnt

# DFP

class DFP():

def \_\_init\_\_(self, x, f, epsilon = 1e-6, max\_iter = 100):

self.x = x

self.f = f

self.epsilon = epsilon

self.max\_iter = max\_iter

self.step\_select\_methods = {

"fixed": {"init\_step": 0.1},

"Newton": {"Hesse\_f": None},

"GoldenCut": {"max\_step": 1},

}

def optimize(self, grad\_f, StepSelectMethod, \*\*kwargs):

if StepSelectMethod not in self.step\_select\_methods:

raise ValueError(f"Unknown step select method: {StepSelectMethod}")

x = self.x

f = self.f

epsilon = self.epsilon

max\_iter = self.max\_iter

n = np.size(x)

g = grad\_f(x)

iter\_cnt = 0

if np.linalg.norm(g) >= epsilon:

if StepSelectMethod == "fixed":

pass

elif StepSelectMethod == "Newton":

Hesse\_f = kwargs["Hesse\_f"]

elif StepSelectMethod == "GoldenCut":

if "max\_step" in kwargs:

max\_step = kwargs["max\_step"]

else:

max\_step = self.step\_select\_methods["GoldenCut"]["max\_step"]

end = False

for \_ in range(max\_iter):

H = np.eye(n)

p = -H@g

iter\_in = 0

while(1):

# find step = argmin\_step f(x + step\*p)

if StepSelectMethod == "fixed":

step = self.step\_select\_methods["fixed"]["init\_step"]

x = x + step\*p

elif StepSelectMethod == "Newton":

# dlambda f(x + lambda\*p) = dot(grad\_f(x + lambda\*p), p)

# ddlambda f(x + lambda\*p) = dot(Hesse\_f(x + lambda\*p)@p, p)

step, x = StepSelect\_Newton(x, grad\_f, Hesse\_f, p, 0)

elif StepSelectMethod == "GoldenCut":

def f\_x\_plus\_lp(l):

return f(x + l\*p)

step = StepSelect\_GoldenCut(f\_x\_plus\_lp, 0, max\_step)

x = x + step\*p

g\_pre = g

g = grad\_f(x)

if np.linalg.norm(g) < epsilon:

end = True

break

iter\_in += 1

if iter\_in == n:

break

delta\_x = step\*p

delta\_g = g - g\_pre

r = H@delta\_g

# np.outer(x, x) is just delta\_x[:, np.newaxis] @ delta\_x[np.newaxis, :]

H += np.outer(delta\_x, delta\_x)/np.dot(delta\_x, delta\_g) - np.outer(r, r)/np.dot(delta\_g, r)

if end == True:

break

iter\_cnt += 1

return x, f(x), iter\_cnt

def f(x):

return 4\*x[0]\*\*2 + 4\*x[1]\*\*2 - 4\*x[0]\*x[1] - 12\*x[1]

def grad\_f(x):

return np.array([8\*x[0] - 4\*x[1], 8\*x[1] - 4\*x[0] - 12])

def Hesse\_f(x):

return np.array([[8, -4], [-4, 8]])

# Attention: in python, vector and matrix is different.

# When use vector (such as variable x), it's better to use vector, x = np.array([1, 1])

# In matrix calculation, if it will not be confused, numpy will decide whether it is row or column vector automatically,

# otherwise, make a special announcement as `x[:, np.newaxis]`

start = time.time()

minpoint, fmin, itertime = SteepestDescent(f, grad\_f, np.array([-1/2, 1]), "fixed", Hesse\_f)

end = time.time()

print(f"Steepest Descent, Step select method: fixed")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

start = time.time()

minpoint, fmin, itertime = SteepestDescent(f, grad\_f, np.array([-1/2, 1]), "Newton", Hesse\_f)

end = time.time()

print(f"Steepest Descent, Step select method: Newton")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

optimizer = ConjugateGradient\_FR(np.array([-1/2, 1]), f)

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, "Newton", Hesse\_f = Hesse\_f)

end = time.time()

print(f"Conjugate Gradient (FR), Step select method: Newton")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

optimizer = DFP(np.array([-1/2, 1]), f)

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, "GoldenCut")

end = time.time()

print(f"DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, "Newton", Hesse\_f = Hesse\_f)

end = time.time()

print(f"DFP, Step select method: Newton")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

## 结果

Steepest Descent, Step select method: fixed

min point: [0.99999999 1.99999999], fmin: -12.0, iter time: 36

run time: 0.0009999275207519531

Steepest Descent, Step select method: Newton

min point: [0.99997635 1.99998423], fmin: -11.999999998260051, iter time: 100

run time: 0.0039997100830078125

Conjugate Gradient (FR), Step select method: Newton

min point: [1. 2.], fmin: -12.0, iter time: 0

run time: 0.0009999275207519531

DFP, Step select method: GoldenCut

min point: [0.99999996 1.99999989], fmin: -11.999999999999964, iter time: 20

run time: 0.017999887466430664

DFP, Step select method: Newton

min point: [0.99997635 1.99998423], fmin: -11.999999998260051, iter time: 100

run time: 0.0070002079010009766

发现：在最速下降法中，精确一维搜索，产生了锯齿现象；Newton法搜索初始点如果不加人为改变可能在后期导致步长一直为零，无法收敛到要求精度；FR共轭梯度法采用精确一维搜索时，收敛速度很快；后两种方法的收敛速度比第一种方法快；最速下降法的步长选择用非精确一维搜索也可以。

1. 约束优化

## 代码

# T5: Exterior & Interior Point

class ConstraintOptimization:

pass

class ExteriorPoint:

# Generally, accuracy of outer process should be lower then inner one, otherwise it can't converge

def \_\_init\_\_(self, x, f, g, h, init\_M = 1, c = 4, epsilon = 1e-3, max\_iter = 100):

self.x = x

self.f = f

self.g = g

self.h = h

self.init\_M = init\_M

self.c = c

self.epsilon = epsilon

self.max\_iter = max\_iter

def p(self, x, g, h):

sum\_g = 0

sum\_h = 0

if g != None:

gs = g(x)

sum\_g = np.sum(np.minimum(gs, np.zeros(np.size(gs)))\*\*2)

if h != None:

sum\_h = np.sum(h(x)\*\*2)

return sum\_g + sum\_h

def grad\_p(self, x, g, h, grad\_g, grad\_h):

# grad\_p = sum(g(x)\*grad\_g - |g(x)|\*grad\_g) + sum(2\*h(x)\*grad\_h))

# = sum(2\*min{g(x), 0}\*grad\_g) + sum(2\*h(x)\*grad\_h))

n = np.size(x)

sum\_g = np.zeros(n)

sum\_h = np.zeros(n)

if g != None:

gx = g(x)

grad\_gx = grad\_g(x)

# here use g(x) is a single-row matrix, we want each column of g(x) multiplies relative column of grad\_g(x)

sum\_g = np.sum(gx\*grad\_gx - np.abs(gx)\*grad\_gx, axis = 1)

if h != None:

hx = h(x)

grad\_hx = grad\_h(x)

sum\_h = np.sum(2\*hx\*grad\_hx, axis = 1)

return sum\_g + sum\_h

def optimize(self, grad\_f, grad\_g, grad\_h):

x = self.x

f = self.f

g = self.g

h = self.h

M = self.init\_M

c = self.c

epsilon = self.epsilon

max\_iter = self.max\_iter

iter\_cnt = 0

for \_ in range(max\_iter):

def f\_plus\_Mp(x):

return f(x) + M\*self.p(x, g, h)

def grad\_f\_plus\_Mp(x):

return grad\_f(x) + M\*self.grad\_p(x, g, h, grad\_g, grad\_h)

optimizer = DFP(x, f\_plus\_Mp)

x, f\_plus\_Mp\_min, \_ = optimizer.optimize(grad\_f\_plus\_Mp, "GoldenCut", max\_step = 0.1)

# must judge error after once optimized, because if init point is in the field, error = 0

error = np.abs(f\_plus\_Mp\_min - f(x))

if error < epsilon:

break

M = c\*M

iter\_cnt += 1

return x, f(x), iter\_cnt

class InteriorPoint:

# accuracy should not be too high, otherwise the point will go out of the boundary

def \_\_init\_\_(self, x, f, g, init\_r = 10, c = 0.1, epsilon = 0.1, max\_iter = 100):

self.x = x

self.f = f

self.g = g

self.init\_r = init\_r

self.c = c

self.epsilon = epsilon

self.max\_iter = max\_iter

self.punish\_functions = {

"reciprocal",

"logarithm",

}

def B\_r(self, x, g):

sum\_g = 0

if g != None:

sum\_g = np.sum(1/g(x))

return sum\_g

def grad\_B\_r(self, x, g, grad\_g):

# grad\_B\_r = sum(-grad\_g(x)/[g(x)]^2)

sum\_g = np.zeros(np.size(x))

if g != None:

sum\_g = np.sum(-1/g(x)\*\*2\*grad\_g(x), axis = 1)

return sum\_g

def B\_l(self, x, g):

sum\_g = 0

if g != None:

sum\_g = -np.sum(np.log(g(x)))

return sum\_g

def grad\_B\_l(self, x, g, grad\_g):

# grad\_B\_l = -sum(grad\_g(x)/g(x))

sum\_g = np.zeros(np.size(x))

if g != None:

sum\_g = -np.sum(1/g(x)\*grad\_g(x), axis = 1)

return sum\_g

def optimize(self, grad\_f, grad\_g, PunishFunction):

x = self.x

f = self.f

g = self.g

r = self.init\_r

c = self.c

epsilon = self.epsilon

max\_iter = self.max\_iter

if PunishFunction not in self.punish\_functions:

raise ValueError(f"Unknown punish function: {PunishFunction}")

if PunishFunction == "reciprocal":

B = self.B\_r

grad\_B = self.grad\_B\_r

elif PunishFunction == "logarithm":

B = self.B\_l

grad\_B = self.grad\_B\_l

iter\_cnt = 0

for \_ in range(max\_iter):

def f\_plus\_rB(x):

return f(x) + r\*B(x, g)

def grad\_f\_plus\_rB(x):

return grad\_f(x) + r\*grad\_B(x, g, grad\_g)

print(f"g:{g(x)}")

print(f"x:{x} f(x):{f(x)} r\*B:{r\*B(x, g)}")

optimizer = DFP(x, f\_plus\_rB)

x, f\_plus\_rB\_min, \_ = optimizer.optimize(grad\_f\_plus\_rB, "GoldenCut")

# must judge error after once optimized, because if init point is in the field, error = 0

error = np.abs(f\_plus\_rB\_min - f(x))

if error < epsilon:

break

r = c\*r

iter\_cnt += 1

return x, f(x), iter\_cnt

# In this case, as the number of constraints can be single or multiple, constraints should represents as single-row matrix

# different dims of x by row, different constraints by column

def f(x):

return x[0]\*\*2 + x[1]\*\*2

def grad\_f(x):

return np.array([2\*x[0], 2\*x[1]])

# remember to add an extra [] even if g(x) has only one formula,

# otherwise it will be regarded as vector but not matrix (eg. in numpy.sum, axis = 1 is not available)

def g(x):

return np.array([[x[0] - 1]])

def grad\_g(x):

return np.array([[1], [0]])

print(f"function 1")

optimizer = ExteriorPoint(np.array([2, 2]), f, g, None)

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, grad\_g, None)

end = time.time()

print(f"Exterior point, non-constraint method: DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

optimizer = InteriorPoint(np.array([2, 2]), f, g)

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, grad\_g, "reciprocal")

end = time.time()

print(f"Interior point, punish function: reciprocal, non-constraint method: DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, grad\_g, "logarithm")

end = time.time()

print(f"Interior point, punish function: logarithm, non-constraint method: DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

def f(x):

return -x[0]\*x[1]

def grad\_f(x):

return np.array([-x[1], -x[0]])

# g is a single-row matrix, but not vector

def g(x):

return np.array([[-x[0] - x[1]\*\*2 + 1, x[0] + x[1]]])

def grad\_g(x):

return np.array([[-1, 1], [-2\*x[1], 1]])

print(f"function 2")

optimizer = ExteriorPoint(np.array([2, 2]), f, g, None)

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, grad\_g, None)

end = time.time()

print(f"Exterior point, non-constraint method: DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

optimizer = InteriorPoint(np.array([0.5, 0.5]), f, g)

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, grad\_g, "reciprocal")

end = time.time()

print(f"Interior point, punish function: reciprocal, non-constraint method: DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

start = time.time()

minpoint, fmin, itertime = optimizer.optimize(grad\_f, grad\_g, "logarithm")

end = time.time()

print(f"Interior point, punish function: logarithm, non-constraint method: DFP, Step select method: GoldenCut")

print(f"min point: {minpoint}, fmin: {fmin}, iter time: {itertime}")

print(f"run time: {end - start}\n")

## 结果

function 1 （默认参数）

Exterior point, non-constraint method: DFP, Step select method: GoldenCut

min point: [9.99024390e-01 3.24611342e-07], fmin: 0.9980497319455207, iter time: 5

run time: 0.08800005912780762

x:[2 2] f(x):8 r\*B:10.0

x:[ 2.43342767e+00 -3.45953032e-09] f(x):5.921570246498887 r\*B:0.6976285011546657

x:[ 1.56519771e+00 -5.22243203e-16] f(x):2.449843873161911 r\*B:0.17692923755796758

x:[5.61227777e-02 3.72204664e-17] f(x):0.003149766175120704 r\*B:-0.010594598284168099

Interior point, punish function: reciprocal, non-constraint method: DFP, Step select method: GoldenCut

min point: [ 5.05089635e-03 -4.13672368e-19], fmin: 2.5511553937870017e-05, iter time: 3

run time: 0.03299999237060547

x:[2 2] f(x):8 r\*B:-0.0

x:[2.79128769e+00 2.49881952e-07] f(x):7.791286966323492 r\*B:-0.5829347409090372

x:[1.36602540e+00 1.17088329e-07] f(x):1.8660253974930239 r\*B:0.10050525450338095

x:[1.04772255e+00 1.05032587e-07] f(x):1.0977225396904087 r\*B:0.030423512673469995

Interior point, punish function: logarithm, non-constraint method: DFP, Step select method: GoldenCut

min point: [1.00497525e+00 1.15884482e-07], fmin: 1.0099752456727895, iter time: 3

run time: 0.0559999942779541

function 2 （调整参数init\_r = 1和精确一维搜索方法的max\_step = 0.01以及epsilon = 0.05）

Exterior point, non-constraint method: DFP, Step select method: GoldenCut

min point: [0.66744612 0.57765251], fmin: -0.3855519289214287, iter time: 4

run time: 0.9879999160766602

x:[0.5 0.5] f(x):-0.25 r\*B:5.0

x:[0.19722759 0.45421414] f(x):-0.08958355938882287 r\*B:0.32116096611014505

x:[0.38249179 0.4528998 ] f(x):-0.17323045618435354 r\*B:0.03621932595352922

x:[0.57234874 0.54021587] f(x):-0.3091918691753714 r\*B:0.00826161419513253

Interior point, punish function: reciprocal, non-constraint method: DFP, Step select method: GoldenCut

min point: [0.63732836 0.56621404], fmin: -0.36086426387137066, iter time: 3

run time: 0.6490001678466797

x:[0.5 0.5] f(x):-0.25 r\*B:1.3862943611198906

x:[0.2608292 0.44877479] f(x):-0.11705356828713369 r\*B:0.09633688469896051

x:[0.5511195 0.53431868] f(x):-0.29447344301899026 r\*B:0.017296679595515816

Interior point, punish function: logarithm, non-constraint method: DFP, Step select method: GoldenCut

min point: [0.65332398 0.57393144], fmin: -0.37496317266498785, iter time: 2

run time: 0.4089999198913574

发现：内点法的终止条件需要设置得宽松一些。内点法对参数比较苛刻，可能会出现不收敛、迭代点跑出可行域等问题，要多调几次参才能计算。多数情况下对数罚函数比倒数罚函数更好用一些。