MDS 6106 – Introduction to Optimization

Exercise Sheet Nr.: 03

Name: Kang Haoyu Student ID: 220041025

In the creation of this solution sheet, I worked together with:

Name: Student ID:

Name: Student ID:

Name: Student ID:

For correction:

Exercise						Σ
Grading						

 # A 3.1

a) In the general section method, I set initial Xl=0, Xr=4,and through the computation by Python, this method goes through 27 iterations to get 10^-5 accuracy.
At the 27th iteration, Xl=1.4275329, Xr=1.42757, X=(Xl+Xr)/2=1.42755, and the optimal solution=-1.775725. The python code to solve this problem is shown as follow

```
from math import *
def f(x):
    return x**2/10-2*sin(x)
def golden_section(function,initial_l,initial_r,tol,theta=0.382):
    """
    :param function: the optimal function we are going to call
    :param initial_l: the initial xl
    :param initial_r: the initial xr
    :param theta: default 0.382
    :param tol: a value which can estimate whether it gets a good iteration
    :return:
    """
    xl=initial_l
    xr=initial_r
    i=1
    while True:
        xl_=theta*xr+(1-theta)*xl
        xr_ = theta * xl + (1 - theta) * xr
        if function(xl_)<function(xr_):
            xr=xr_
        else:
            xl = xl_
        if (xr-xl)<tol:
          print("iteration{}:xl={}, xr={}, x={},f(x) is 
{}".format(i,xl,xr,(xr+xl)/2,function((xr+xl)/2)))
          return (xr+xl)/2
        print("iteration{}:xl={}, xr={}, f(x) is {}".format(i,xl,xr,function((xr+xl)/2)))
        i+=1
golden_section(f,0,4,0.00001)
```

b) When using golden section method, I set initial Xl=0, Xr=1,and go through 24 times iterations, it get 10^-5 accuracy, with the result of Xl=0.5885001, Xr=0.5885409, thus optimal solution X=(Xl+Xr)/2
When using bisection method, I also set initial Xl=0 where gradient of g(x)<0. Set Xr=1 where gradient of g(x)>0. Then it goes through 17 times iteration with the result of Xl=0.588531, Xr=0.588562. The optimal solution X= (Xr+Xl)/2=0.588546, and optimal value g(x)=-0.27661.
Conclusion: with golden section, it goes through with 24 times iteration. However, with bisection method, it goes through with 17 times iteration. The python code to solve this problem

is shown as follow.

```
def g(x):
    return 1/e**x-cos(x)
def g_(x):
    return -1/e**x+sin(x)
def Bisection(initial_l,initial_r,tol):
    xl=initial_l
    xr=initial_r
    i=1
    while True:
        xm = (xr + xl) / 2
        if g_(xm)==0:
            return xm
        if g_(xm)>0:
            xr=xm
        else:
            xl=xm
        if abs(xr-xl)<tol:
            print("iteration{}:xl={}, xr={}, x={},f(x) is {}".format(i, xl, xr,(xr + xl) /
2, g((xr + xl) / 2)))
            return (xr+xl)/2
        print("iteration{}:xl={}, xr={}, f(x) is {}".format(i,xl,xr,g((xr+xl)/2)))
        i += 1
golden_section(g,0,1,0.00001)
Bisection(0,1,0.00001)
```

A. 3. 2

a) In order to verify d is a descent direction of f at x, we have to verify $\nabla f(x)^T d < 0$.

$$\nabla f(x)^T d = -\nabla f(x)^T \frac{\partial f}{\partial x_i}(x) \cdot e_j$$

$$= -\left(\frac{\partial f}{\partial x_1} \ \frac{\partial f}{\partial x_2} \ \cdots \ \frac{\partial f}{\partial x_n}\right)\begin{pmatrix} 0 \\ \vdots \\ \frac{\partial f}{\partial x_j} \\ \vdots \\ 0 \end{pmatrix}$$

$$= -\left(\frac{\partial f}{\partial x_j}\right)^2$$

b) We can derive $\nabla f(x)^T d = -\frac{1}{\sqrt{\varepsilon + \|\nabla f(x)\|^2}} f(x)^T \nabla f(x) = -\frac{1}{\sqrt{\varepsilon + \|\nabla f(x)\|^2}} \|\nabla f(x)\|^2 < 0$

Thus, we verified that d is a descent direction of f at x

c) Because we know $\nabla^2 f(x)$ is positive definite, so we have

$$h^T \nabla^2 f(x) h > 0, \quad \forall h \in \mathbb{R}^n$$

We set $e_i \in \mathbb{R}^n$ is the i-th unit vector and $i \in \{1, \cdots, n\}$. So we have

$$e_i^T \nabla^2 f(x) e_i = \nabla^2 f(x)_{ii} > 0$$

Thus, we don't divided by zero because $\nabla^2 f(x)_{ii} > 0$

For the direction d, we can derive

$$\nabla f(x)^T d = -\left(\frac{\partial f}{\partial x_1} \; \frac{\partial f}{\partial x_2} \cdots \frac{\partial f}{\partial x_n}\right) \begin{pmatrix} \nabla f(x)_1 / \nabla^2 f(x)_{11} \\ \vdots \\ \nabla f(x)_i / \nabla^2 f(x)_{ii} \\ \vdots \\ \nabla f(x)_n / \nabla^2 f(x)_{nn} \end{pmatrix} = -\sum_{i=1}^{n} \frac{(\nabla f(x)_i)^2}{\nabla^2 f(x)_{ii}} < 0$$

Thus, we have verified that d is a descent direction of f at x.

A 3.3

(a)

- **Part I:**

By solving the equation ∇f(x) = 0, I derive the stationary points as follows:

A(-7, -1) B(-3, 0) C(-1, 1) D(-2/9(15+4√3), -1/√3) E(-2/9(-15+4√3), 1/√3)

By calculating the Hessian matrix for point A,B,C,D,E, we can find at points A,B,C, the Hessian matrix is positive definite, and at points D,E, the Hessian matrix is indefinite. Thus, points A, B, C are strict local minimizer, and points D,E are saddle points.

We can calculate the value as f(A)=f(B)=f(C)=0. Because f(x)=$f_1(x)^2$+$f_2(x)^2$, so f(x)>=0, and it is continuous and differentiable. So points A, B, C are all global minimizer, but not strict.

Thus, points A, B, C are strict local minimizer and global minimizer(not strict). Points D,E are saddle points.

- **Part II: The gradient method**

By applying different step size strategies(Backstracking, Exact line search and diminishing step size method) in this problem, we set the initial point (0,0) according to the requestion. I find that all converges to (-1,1) approximately. The number of iterations with the different step size are 162(Backtracking), 11993(Diminishing step size), and 7(Exact line search). Thus we can get that the Exact line search get a better effect and rate of convergence at this condition. The python code is as follow.

```
from math import *
# from A3_1 import *
def f1(x1,x2):
    return 3 + x1 + ((1 - x2) * x2 - 2) * x2
def f2(x1,x2):
    return 3 + x1 + (x2 - 3) * x2
def f(f1,f2,x1,x2):
    return f1(x1,x2)**2+f2(x1,x2)**2
def gradient(f1,f2,x1,x2):
    grad=[]
    grad.append(2*f1(x1,x2)+2*f2(x1,x2))
    grad.append(2*f1(x1,x2)*((2*x2)-3*(x2**2)-2)+2*f2(x1,x2)*(2*x2-3))
    return grad
def func_alpha(f1,f2,x1,x2,alpha,dire):
    return f1(x1+alpha*dire[0], x2+alpha*dire[1]) ** 2 + f2(x1+alpha*dire[0],
x2++alpha*dire[1]) ** 2
def golden_section(function,x1,x2,dire,initial_l,initial_r,tol,theta=0.382):
    """


    :param function: the optimal function we are going to call
    :param initial_l: the initial xl
    :param initial_r: the initial xr
    :param theta: default 0.382
    :param tol: a value which can estimate whether it gets a good iteration
```

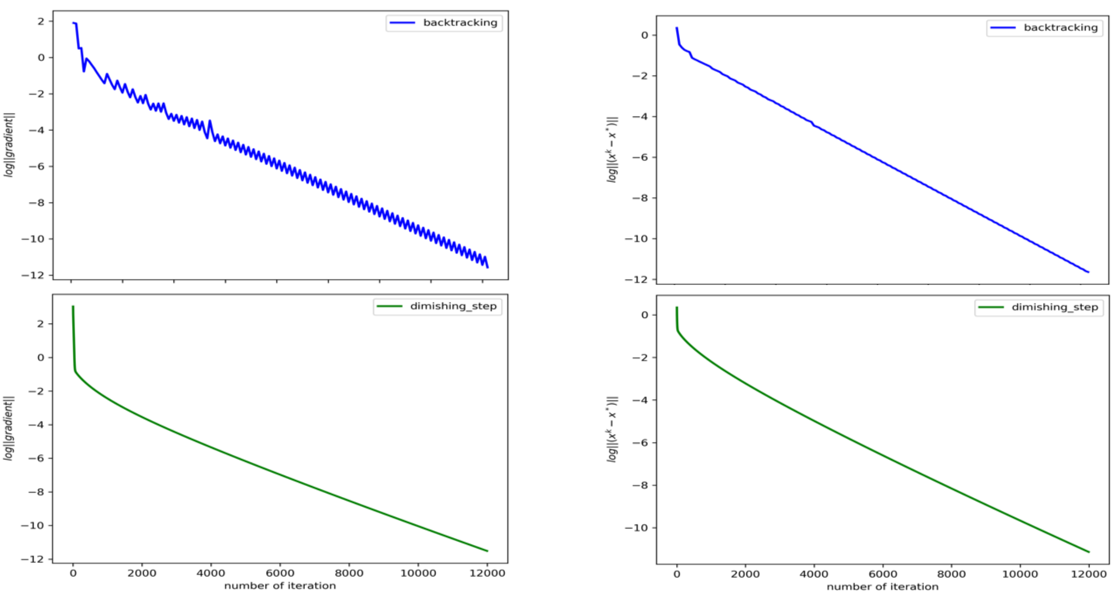
```
    :return:
    """
    xl=initial_l
    xr=initial_r
    i=1
    while True:
        xl_=theta*xr+(1-theta)*xl
        xr_ = theta * xl + (1 - theta) * xr
        if function(f1,f2,x1,x2,xl_,dire)<function(f1,f2,x1,x2,xr_,dire):
            xr=xr_
        else:
            xl = xl_
        if (xr-xl)<tol:
            # print("iteration{}:xl={}, xr={}, x={},f(x) is
{}".format(i,xl,xr,(xr+xl)/2,function(f1,f2,x1,x2,(xr+xl)/2,dire)))
            return (xr+xl)/2
        # print("iteration{}:xl={}, xr={}, f(x) is
{}".format(i,xl,xr,function(f1,f2,x1,x2,(xr+xl)/2,dire)))
        i+=1
def direction(grad):
    dire=[]
    dire.append(-grad[0])
    dire.append(-grad[1])
    return dire
def backtracking(sigma,gama,tol,initial_point):
    ir=1
    save_xk=[]
    save_norm_grad=[]
    x1=initial_point[0]
    x2=initial_point[1]
    save_xk.append((x1,x2))
    grad=gradient(f1,f2,x1,x2)
    print(grad)
    while True:
        alpha = 1
        dire = direction(grad)
        while f(f1,f2,x1+alpha*dire[0],x2+alpha*dire[1])-
f(f1,f2,x1,x2)>gama*alpha*(grad[0]*dire[0]+grad[1]*dire[1]):
            alpha=alpha*sigma
        x1=x1+alpha*dire[0]
        x2=x2+alpha*dire[1]
        function_value=f(f1,f2,x1,x2)
        grad = gradient(f1, f2, x1, x2)
        norm_grad = (grad[0] ** 2 + grad[1] ** 2)**0.5
```

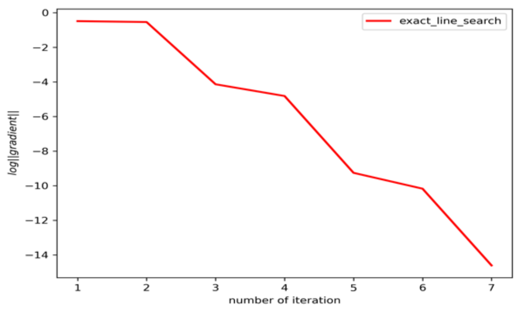
```
        print("iteration{}:xk={},norm_grad={},f(x) =
{}".format(ir,[x1,x2],norm_grad,function_value))
        save_xk.append((x1,x2))
        save_norm_grad.append(norm_grad)
        if norm_grad<tol:
            break
        ir += 1
    return save_xk,save_norm_grad,ir

# backtracking(0.5,0.1,1e-5,[0,0])
# grad=gradient(f1,f2,0,0)
# dire=direction(grad)
# print(golden_section(func_alpha,0,0,dire,0,2,1e-6,theta=0.382))

def exact_line_search(tol,initial_point):
    ir = 1
    save_xk=[]
    save_norm_grad=[]
    x1 = initial_point[0]
    x2 = initial_point[1]
    save_xk.append((x1,x2))
    grad=gradient(f1,f2,x1,x2)
    # print(grad)
    while True:
        dire = direction(grad)
        alpha=golden_section(func_alpha,x1,x2,dire,0,2,1e-6,theta=0.382)
        x1 = x1 + alpha * dire[0]
        x2 = x2 + alpha * dire[1]
        function_value = f(f1, f2, x1, x2)
        grad = gradient(f1, f2, x1, x2)
        norm_grad = (grad[0] ** 2 + grad[1] ** 2) ** 0.5
        print("iteration{}:xk={},norm_grad={},f(x) = {}".format(ir, [x1, x2], norm_grad,
function_value))
        save_xk.append((x1,x2))
        save_norm_grad.append(norm_grad)
        if norm_grad < tol:
            break
        ir += 1
    return save_xk, save_norm_grad, ir
# exact_line_search(1e-5,[0,0])

def dimishing_step(tol,initial_point):
    ir = 1
    save_xk = []
```

```
    save_norm_grad = []
    x1 = initial_point[0]
    x2 = initial_point[1]
    save_xk.append((x1, x2))
    grad = gradient(f1, f2, x1, x2)
    while True:
        dire = direction(grad)
        alpha=0.01/log(ir+15)
        x1 = x1 + alpha * dire[0]
        x2 = x2 + alpha * dire[1]
        function_value = f(f1, f2, x1, x2)
        grad = gradient(f1, f2, x1, x2)
        norm_grad = (grad[0] ** 2 + grad[1] ** 2) ** 0.5
        print("iteration{}:xk={},norm_grad={},f(x) = {}".format(ir, [x1, x2], norm_grad,
function_value))
        save_xk.append((x1, x2))
        save_norm_grad.append(norm_grad)
        if norm_grad < tol:
            break
        ir += 1
    return save_xk, save_norm_grad, ir
backtracking(0.5,0.1,1e-5,[0,0])
exact_line_search(1e-5,[0,0])
dimishing_step(1e-5,[0,0])
```

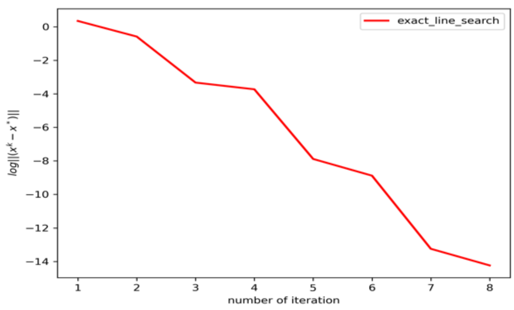
(b)

According to previous question, The limit points of the three sequences we have generated with different step size are (-1,1), and then I draw the figures of the$(\| \nabla f(x^k) \|)_k$ and $(\| x^k - x^* \|)_k$ in Figure 1 as follow





(a) $\log\left(\| \nabla f(x^k) \|\right)_k$



(b) $\log\left(\| x^k - x^* \|\right)_k$

Figure 1

The python code is as follow

```
from A3_3 import *
import matplotlib.pyplot as plt
xk_list_1,norm_grad_1,num_iteration_1=backtracking(0.5,0.1,1e-5,[0,0])
xk_list_2,norm_grad_2,num_iteration_2=exact_line_search(1e-5,[0,0])
xk_list_3,norm_grad_3,num_iteration_3=dimishing_step(1e-5,[0,0])
xk_list=[xk_list_1,xk_list_2,xk_list_3]
norm_grad_list=[norm_grad_1,norm_grad_2,norm_grad_3]
num_iteration_list=[num_iteration_1,num_iteration_2,num_iteration_3]
method_list=['backtracking','exact_line_search','dimishing_step']
color_list=['blue','red','green']
def gradient_plot(num_iteration,norm_grad,method,color):
    iteration = list(i for i in range(1, num_iteration + 1))
    norm_grad = [log(i) for i in norm_grad]
    # plt.figure(1,figsize=(8,10))
    plt.plot(iteration,norm_grad,label=method,color=color,linewidth=2)
    plt.xlabel('number of iteration')
    plt.ylabel('$log||gradient||$')
    plt.legend()
    plt.tight_layout()
    # plt.show()
    plt.savefig(method+"_gradient",dpi=300)
    plt.close()
def norm2(xk,x_star):
    return (xk[0]-x_star[0])**2+(xk[1]-x_star[1])**2
def xk_plot(num_iteration,xk_li,method,color):
    iteration = list(i for i in range(1, num_iteration+2))
    x_star=(-1,1)
    xk = [log(norm2(j,x_star)**0.5) for j in xk_li]
    plt.plot(iteration,xk,label=method,color=color,linewidth=2)
    plt.xlabel('number of iteration')
    plt.ylabel('$log||(x^k-x^*)||$')
```

```
    plt.legend()
    plt.tight_layout()
    # plt.show()
    plt.savefig(method+"_xk",dpi=300)
    plt.close()
for i in range(3):
    num_iteration=num_iteration_list[i]
    norm_grad=norm_grad_list[i]
    method=method_list[i]
    color=color_list[i]
    gradient_plot(num_iteration,norm_grad,method,color)
for i in range(3):
    num_iteration=num_iteration_list[i]
    xk_li=xk_list[i]
    method=method_list[i]
    color=color_list[i]
    xk_plot(num_iteration,xk_li,method,color)
```

(c)

In this problem, I choose 10 initial points (-10,-2), (-2,2), (-10,2), (5,-2), (2,2), (10,2), (-5,-2), (-5,2), (-2,-2), (5,-2). In order to guarantee convergence of the dilimishing step size menthod, I adjust the $\partial_k = 0.01/log(k+12)$. The illustration of convergence path of different gradient method is shown in Figure 2,Figure 3and Figure 4, which are respectively for backtracking, diminishing step size and exact line search.

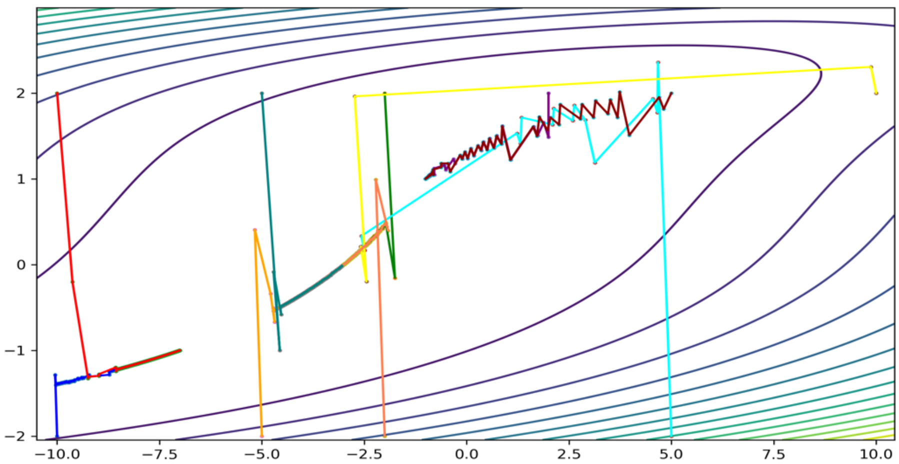


Figure 2 Path of backtracking

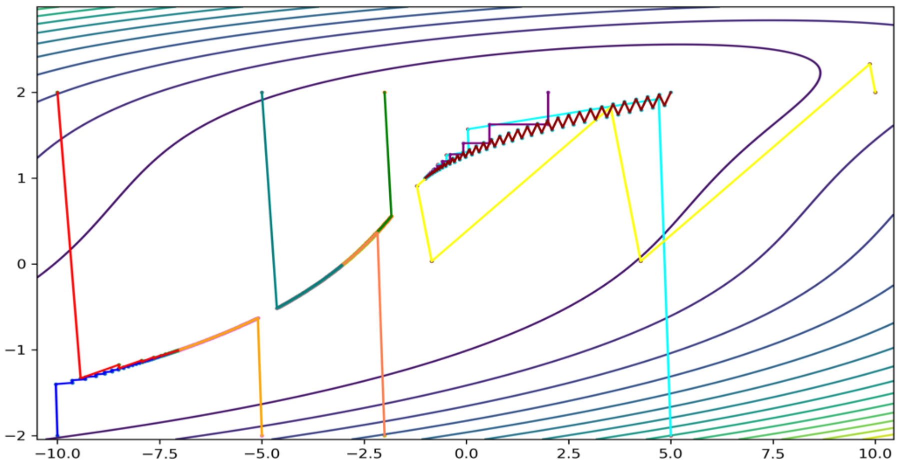


Figure 3 Path of diminishing step size

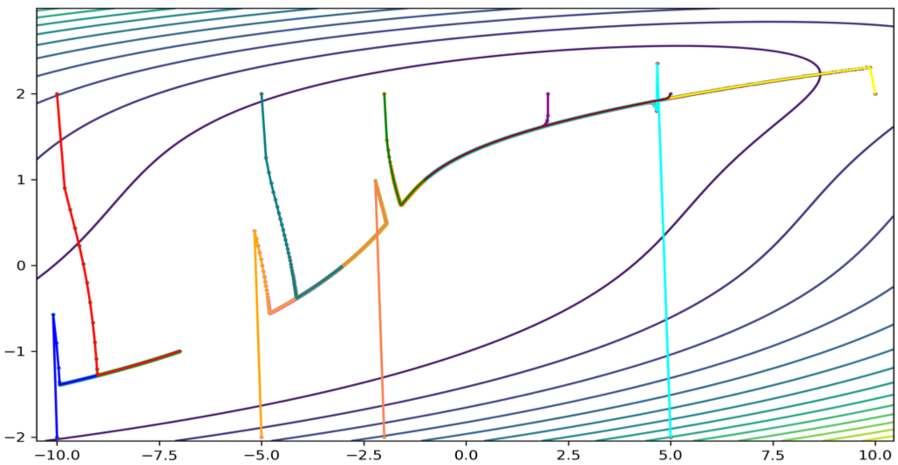


Figure 4 Path of exact line search

The python code for these pictures is shown as follow.

```
from A3_3 import *
import numpy as np
import matplotlib.pyplot as plt
initial_points=[[-10,-2],[-2,2],[-10,2],[5,-2],[2,2],[10,2],[-5,-2],[-5,2],[-2,-2],[5,2]]
#建立步长为0.01，即每隔0.01取一个点
color_list=['blue','green','red','cyan','purple','yellow','orange','teal','coral','darkre
d']
method_list=['backtracking','exact_line_search','dimishing_step']
def contour_plot():
    step1 = 0.05
    step2=0.01125
```

```
    x1 = np.arange(-10.5,10.5,step1)
    x2 = np.arange(-2.045,3,step2)
    #也可以用 x = np.linspace(-10,10,100)表示从-10 到 10，分 100 份

    #将原始数据变成网格数据形式
    X1,X2 = np.meshgrid(x1,x2)
    #写入函数
    fx=(3+X1+((1-X2)*X2-2)*X2)**2+(3+X1+(X2-3)*X2)**2
    #设置打开画布大小,长 10，宽 6
    plt.figure(figsize=(10,6))
    #填充颜色，f 即 filled
    # plt.contourf(X1,X2,fx)
    #画等高线
    plt.contour(X1,X2,fx,15)
    # plt.show()
def path_plot(initial_point,color,method):
    # contour_plot()
    if method=='backtracking':
        xk_list_1, norm_grad_1, num_iteration_1 = backtracking(0.5, 0.1, 1e-5,
initial_point)
    if method=='exact_line_search':
        xk_list_1, norm_grad_1, num_iteration_1 = exact_line_search(1e-5,initial_point)
    if method=='dimishing_step':
        xk_list_1, norm_grad_1, num_iteration_1 = dimishing_step(1e-5,initial_point)
    x1_list=[xk[0] for xk in xk_list_1]
    x2_list=[xk[1] for xk in xk_list_1]
    plt.plot(x1_list,x2_list,linewidth=1.5,color=color)
    plt.scatter(x1_list,x2_list,s=3)
    # plt.show()
def plot_different_method():
    for method in method_list:
        contour_plot()
        for i in range(10):
            path_plot(initial_points[i],color_list[i],method)
        # plt.show()
        plt.savefig(method + "_contour", dpi=300)
        plt.close()

plot_different_method()
```