**# A 3.1**

1. 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) |

1. 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 3.3**

**(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+), -1/) E(-2/9(-15+), 1/)

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)=+, so f(x)>=0, and it is continuous and differentiable. So points A, B, C are all global minimizer, but not strict.

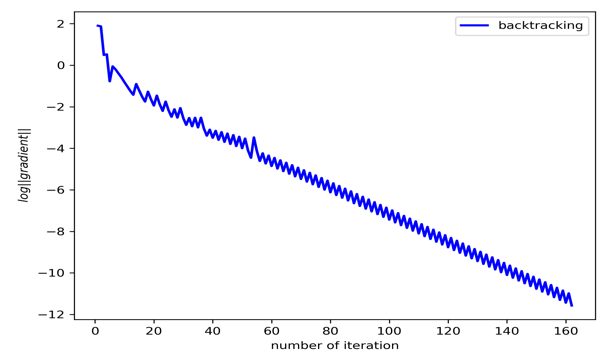
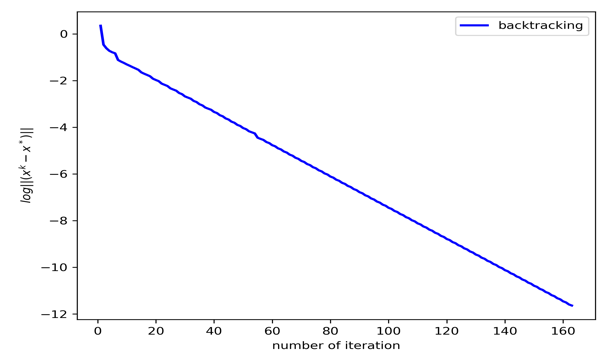
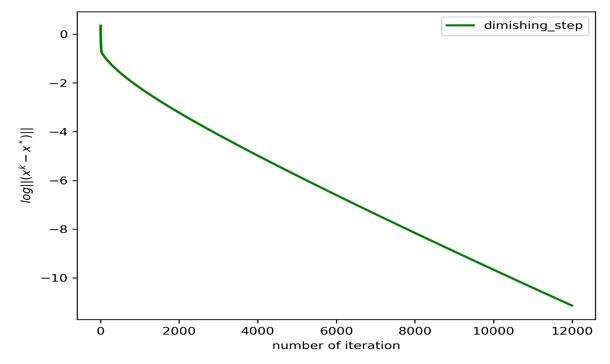
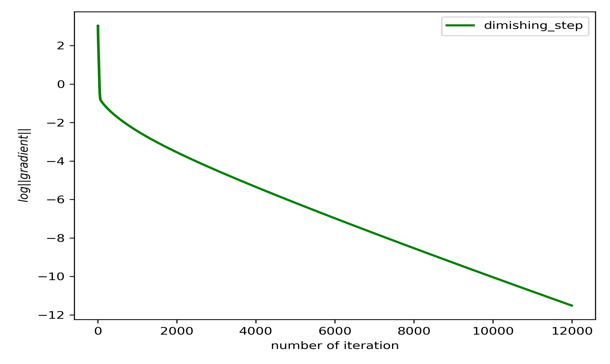
Thus, points A, B, C are strict local minimizer and global minimizer(not strict). Points D,E are saddle points.

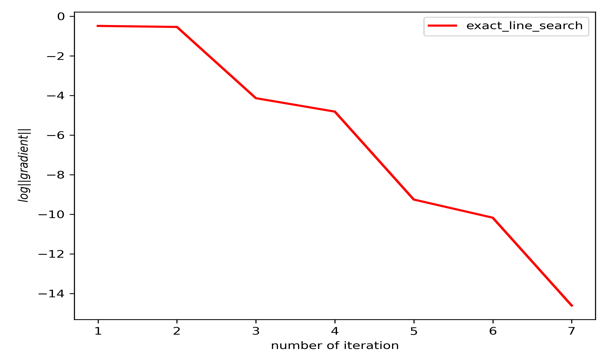
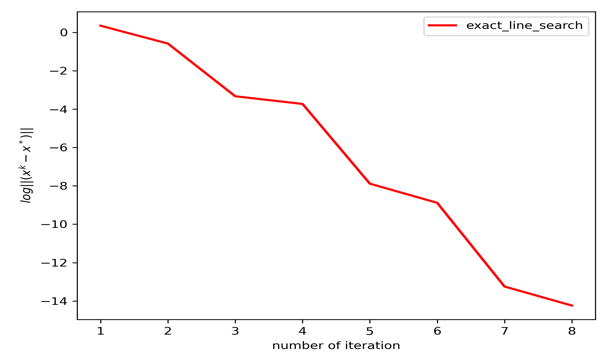
* **Part II: The gradient method**

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 and in Figure 1 as follow



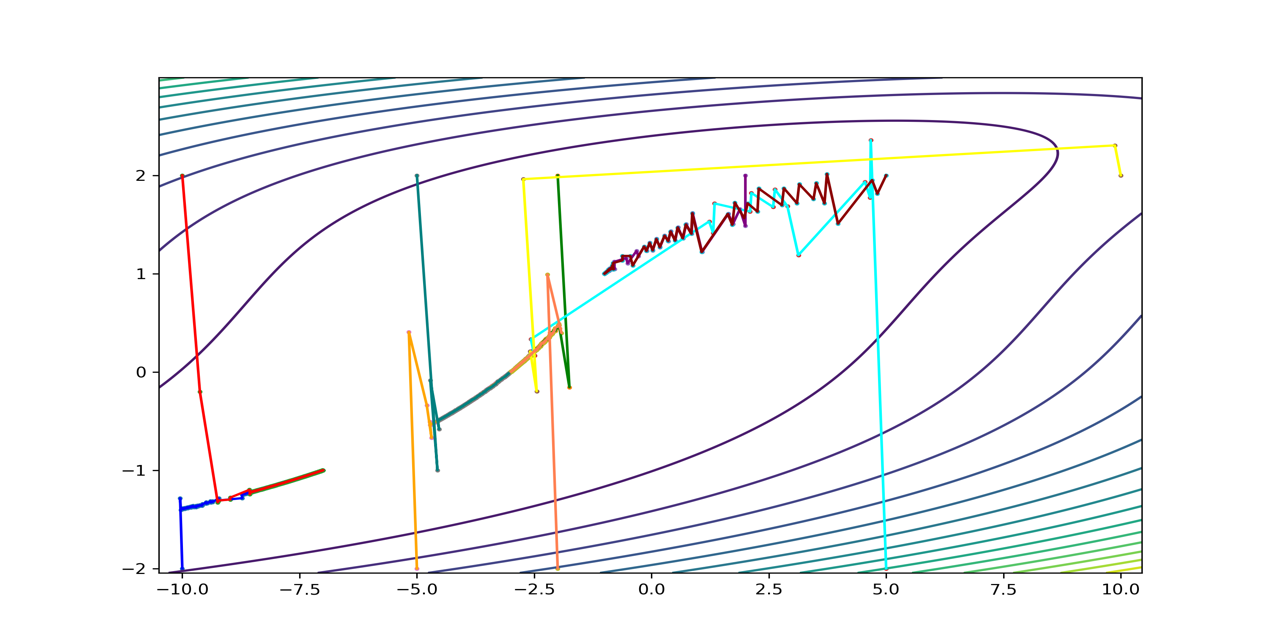
1. log (b) log

**Figure 1**

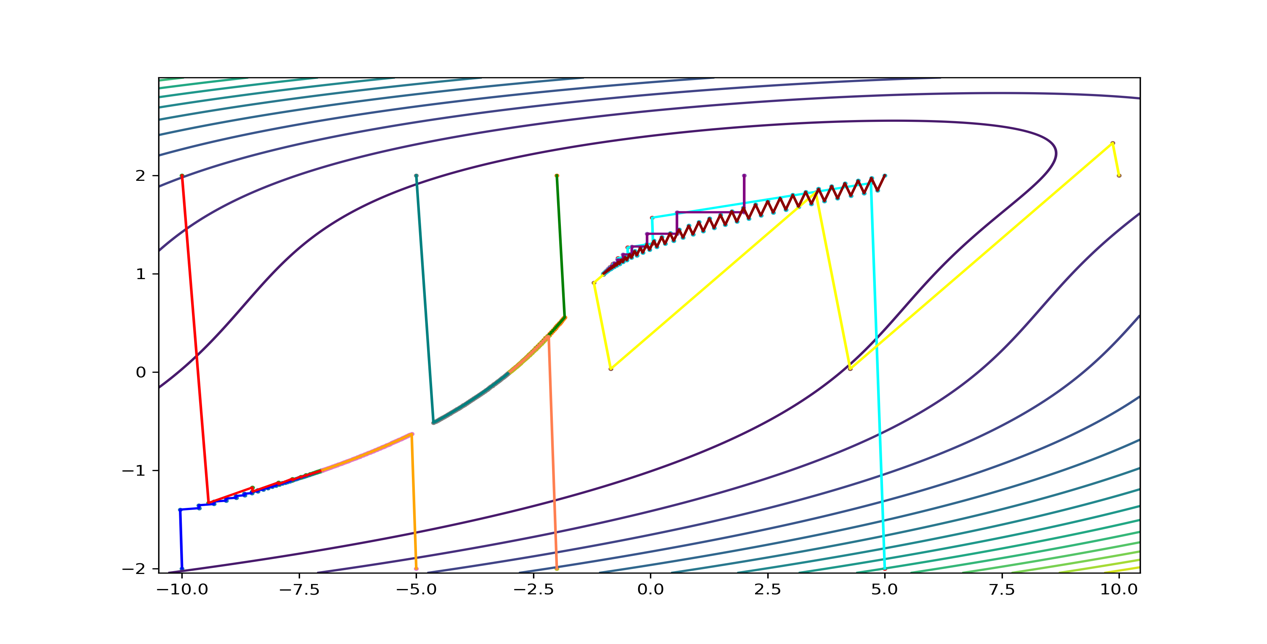
The python code is as follow

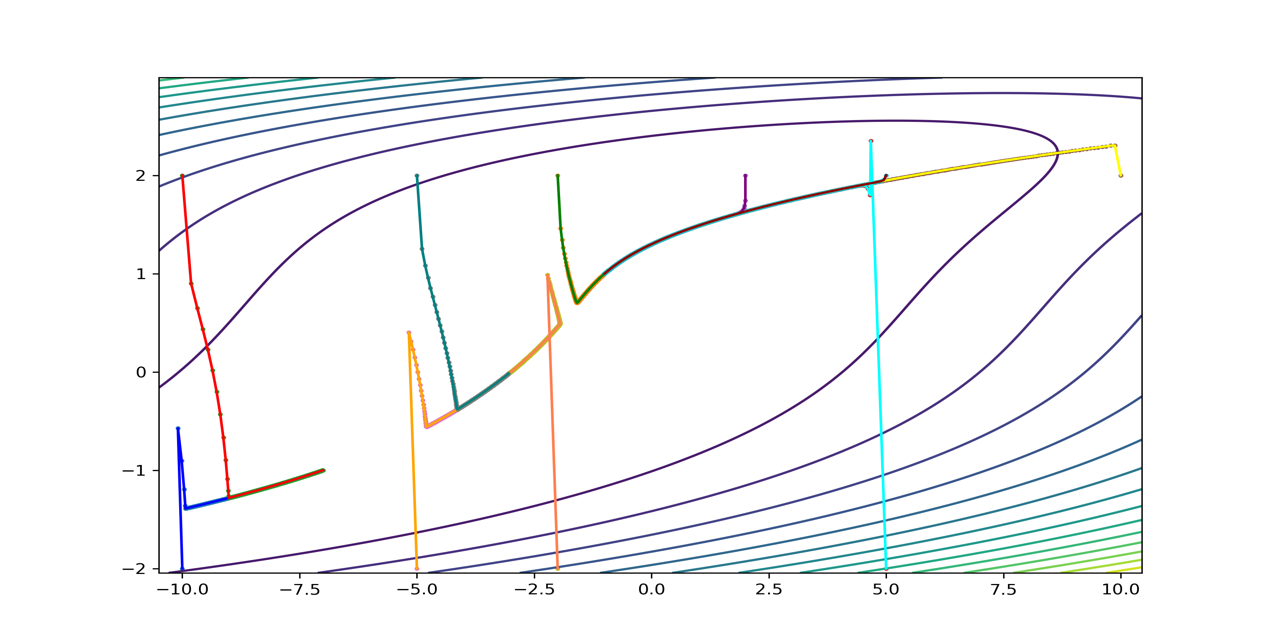
|  |
| --- |
| **from** A3\_3 **import** \* **import** matplotlib.pyplot **as** plt xk\_list\_1,norm\_grad\_1,num\_iteration\_1=backtracking(0.5,0.1,1e-5,[0,0]) xk\_list\_2,norm\_grad\_2,num\_iteration\_2=exact\_line\_search(1e-5,[0,0]) xk\_list\_3,norm\_grad\_3,num\_iteration\_3=dimishing\_step(1e-5,[0,0]) 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 . 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'**,**'darkred'**] 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() |