# Introduction to Optimization: Homework #3

Due on November 20, 2020

*Professor Andre Milzarek*

**Peng Deng**

# Assignment A3.1

In this first exercise, we investigate the performance of the bisection and golden section method.

a) We consider the optimization problem

$$\min_x f(x) := \frac{x^2}{10} - 2\sin(x) \quad \text{s.t.} \quad x \in [0, 4]. \tag{1}$$

Implement the golden section method to solve this problem and output a solution with accuracy at least $10^{-5}$.

b) Consider the minimization problem

$$\min_{x \in \mathbb{R}} g(x) \quad \text{s.t.} \quad x \in [0, 1],$$

where $g$ is given by $g(x) := e^{-x} - \cos(x)$. Solve this problem using the bisection and the golden section method. Compare the number of iterations required to recover a solution in [0,1] with accuracy less or equal than $10^{-5}$

## Solution

### Subproblem (a)

By implementing the golden section method, we can obtain the solution as $x = 1.42755$ after 27 iterations. The python code to solve this problem is showing in as follow.

```python
import numpy as np
# f(x) = x^2/10 - 2*sin(x)
def f(x):
  return x**2/10 - 2*np.sin(x)
xl = 0
xr = 4
fi = (3 - np.sqrt(5)) / 2
gap = 1e-5
num_iteration = 0
last_update = 0 #0:left; 1:right
while (xr - xl) >= gap:
  if num_iteration == 0:
    new_xl = fi * xr + (1 - fi) * xl
    new_xr = (1 - fi) * xr + fi * xl
  elif last_update == 0:
    new_xl = new_xr
    new_xr = (1 - fi) * xr + fi * xl
  else:
    new_xr = new_xl
    new_xl = fi * xr + (1 - fi) * xl

  f_xl = f(new_xl)
  f_xr = f(new_xr)

  if f_xl < f_xr:
    xr = new_xr
    last_update = 1
  else:
    xl = new_xl
    last_update = 0
  num_iteration = num_iteration + 1

ans = (xl + xr) / 2
print('x=', ans)
print('number of iterations:', num_iteration)
```

**Subproblem (b)**

◦ By implementing the golden section method, we can obtain the solution as $x = 0.58853$ after 24 iterations. By implementing the bisection method, we can obtain the solution as $x = 0.58853$ after 17 iterations. The python code to solve this problem is showing as follow.

```
import numpy as np
def f(x):
  return np.e**(-x) - np.cos(x)
def df(x):
  return -np.e**(-x) + np.sin(x)
xl = 0
xr = 1
gap = 1e-5

#golden section
fi = (3 - np.sqrt(5)) / 2
num_iteration = 0
last_update = 0 #0:left; 1:right
while (xr - xl) >= gap:
  if num_iteration == 0:
    new_xl = fi * xr + (1 - fi) * xl
    new_xr = (1 - fi) * xr + fi * xl
  elif last_update == 0:
    new_xl = new_xr
    new_xr = (1 - fi) * xr + fi * xl
  else:
    new_xr = new_xl
    new_xl = fi * xr + (1 - fi) * xl

  f_xl = f(new_xl)
  f_xr = f(new_xr)

  if f_xl < f_xr:
    xr = new_xr
    last_update = 1
  else:
    xl = new_xl
    last_update = 0
  num_iteration = num_iteration + 1

ans = (xl + xr) / 2
print('------golden section---------')
print('x=', ans)
print('number of iterations:', num_iteration)

# bisection
num_iteration = 0
xl = 0
xr = 1
while (xr - xl) >= gap:
  xm = (xl + xr) / 2
  if df(xm) == 0:
    break
  if df(xm) > 0:
    xr = xm
  else:
    xl = xm
  num_iteration = num_iteration + 1
print('------bisection---------')
print('x=', ans)
print('number of iterations:', num_iteration)
```

# Assignment A3.2

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a continuously differentiable function and consider $x \in \mathbb{R}^n$ with $\nabla f(x) \neq 0$. Verify the following statements:

a) Set $d = -(\nabla f(x)_j) \cdot e_j = -\frac{\partial f}{\partial x_j}(x) \cdot e_j$, where $e_j \in \mathbb{R}^n$ is the $j$ -th unit vector and $j \in \{1, \ldots, n\}$ is an index satisfying

$$\left|\frac{\partial f}{\partial x_j}(x)\right| = \max_{1 \leq i \leq n} \left|\frac{\partial f}{\partial x_i}(x)\right| = \|\nabla f(x)\|_\infty$$

Then $d$ is a descent direction of $f$ at $x$.

b) Let us define $d = -\nabla f(x)/\sqrt{\varepsilon + \|\nabla f(x)\|^2}$. Show that $d$ is a descent direction of $f$ at $x$.

c) Suppose that $f$ is twice continuously differentiable and define $d_i = -(\nabla f(x)_i) / \left(\nabla^2 f(x)_{ii}\right)$ for all $i \in \{1, \ldots, n\}$. If $\nabla^2 f(x)$ is positive definite, then $d$ is well-defined (we do not divide by zero) and it is a descent direction of $f$ at $x$.

## Solution

**Subproblem (a)**

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

$$\begin{aligned}
\nabla f(x)^\top d &= -\nabla f(x)^\top \frac{\partial f}{\partial x_j}(x) \cdot e_j \\
&= -\begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \cdots \frac{\partial f}{\partial x_n} \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ \frac{\partial f}{\partial x_j} \\ \vdots \\ 0 \end{pmatrix} \\
&= -\left(\frac{\partial f}{\partial x_j}\right)^2
\end{aligned} \tag{2}$$

Because we know $\nabla f(x) \neq 0$, and $j \in \{1, \ldots, n\}$ is an index satisfying

$$\left|\frac{\partial f}{\partial x_j}(x)\right| = \max_{1 \leq i \leq n} \left|\frac{\partial f}{\partial x_i}(x)\right| = \|\nabla f(x)\|_\infty$$

So we can conclude that $\frac{\partial f}{\partial x_j} \neq 0$, because if $\frac{\partial f}{\partial x_j} = 0$, then for $\forall i$, $\frac{\partial f}{\partial x_i} = 0$, so the gradient will be 0, which is an contradiction. Thus, $\frac{\partial f}{\partial x_j} \neq 0$, and $-\left(\frac{\partial f}{\partial x_j}\right)^2 < 0$. Then, $\nabla f(x)^\top d < 0$, which means $d$ is a descent direction of $f$ at $x$.

**Subproblem (b)**

We can derive

$$\begin{aligned}
\nabla f(x)^\top d &= -\frac{1}{\sqrt{\varepsilon + \|\nabla f(x)\|^2}} f(x)^\top \nabla f(x) \\
&= -\frac{1}{\sqrt{\varepsilon + \|\nabla f(x)\|^2}} \|\nabla f(x)\|^2 < 0
\end{aligned} \tag{3}$$

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

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

$$h^\top \nabla^2 f(x) h > 0, \quad \forall h \in \mathbb{R}^n \tag{4}$$

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

$$e_i^\top \nabla^2 f(x) e_i = \nabla^2 f(x)_{ii} > 0 \tag{5}$$

Thus, we do not divided by zero because $\nabla^2 f(x)_{ii} > 0$.
∘ For the direction $d$, we can derive

$$\begin{aligned} \nabla f(x)^\top d &= -\begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \cdots \frac{\partial f}{\partial x_n} \end{pmatrix} \begin{pmatrix} \nabla f(x)_1 / \nabla^2 f(x)_{11} \\ \vdots \\ \nabla f(x)_i / \nabla^2 f(x)_{ii} \\ \vdots \\ \nabla f(x)_n / \nabla^2 f(x)_{nn} \end{pmatrix} \\ &= -\sum_{i=1}^{n} \frac{(\nabla f(x)_i)^2}{\nabla^2 f(x)_{ii}} < 0 \qquad \text{(Because: } \nabla f(x) \neq 0) \end{aligned} \tag{6}$$

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

# Assignment A3.3

Implement the gradient descent method (Lecture L-06, slide 23) that was presented in the lecture as a function gradient_method in **MATLAB** or **Python**.

The following input functions and parameters should be considered:

- obj, grad - function handles that calculate and return the objective function $f(x)$ and the gradient $\nabla f(x)$ at an input vector $x \in \mathbb{R}^n$. You can treat these handles as functions or fields of a class or structure f or use them directly as input. (For example, your function can have the form gradient_method (obj, grad,....)).
- $x^0-$ the initial point.
- tol - a tolerance parameter. The method should stop whenever the current iterate $x^k$ satisfies the criterion $\left\|\nabla f\left(x^k\right)\right\| \leq$ tol.

We want to analyze the performance of the gradient method for different step size strategies. In particular, we want to test and compare backtracking, exact line search, and diminishing step sizes. The following parameters will be relevant for these strategies:

- $s > 0, \sigma, \gamma \in (0, 1)-$ parameters for backtracking and the Armijo condition.
- alpha - a function that returns a pre-defined, diminishing step size $\alpha_k =$ alpha (k) satisfying $\alpha_k \to 0$ and $\sum \alpha_k = \infty$
- You can use the golden section method from Assignment A3.1 to determine the exact step size $\alpha_k$. The parameters for the golden section method are: maxit (maximum number of iterations), tol (stopping tolerance), [0, a] (the interval of the step size).

You can organize the latter parameters in an appropriate options class or structure. It is also possible to implement two separate algorithms for backtracking and diminishing step sizes. The method(s) should return the final iterate $x^k$ that satisfies the stopping criterion.

a) Test your implementation on the following problem:

$$\min_{x\in\mathbb{R}^2} f(x) = f_1(x)^2 + f_2(x)^2$$

where $f_1 : \mathbb{R}^2 \to \mathbb{R}$ and $f_2 : \mathbb{R}^2 \to \mathbb{R}$ are given by:

$$f_1(x) := 3 + x_1 + ((1 - x_2)\, x_2 - 2)\, x_2$$
$$f_2(x) := 3 + x_1 + (x_2 - 3)\, x_2$$

- Calculate all stationary points of the function $f$ and analyze which of the points are (global/local) minimizer, (global/local) maximizer, or saddle points.

**Hint**: This problem has five stationary points.

- Apply the gradient method with backtracking and parameters $(s, \sigma, \gamma) = (1, 0.5, 0.1)$ with diminishing step sizes

$$\alpha_k = \frac{0.01}{\log(k+2)}$$

and exact line search (maxit $= 100$, tol $= 10^{-6}$, a $= 2$ ) to solve the problem $\min_x f(x)$.

The algorithms should use the stopping tolerance tol $= 10^{-5}$. Test the methods using the initial point $(0, 0)^\top$ and report the behavior and performance of the methods. In particular, for each of the initial points, compare the number of iterations and the point to which each algorithm converged.

b) Adjust your code such that the norm of the gradients, $\left\|\nabla f\left(x^k\right)\right\|$, are saved and returned. Plot the sequences $\left(\left\|\nabla f\left(x^k\right)\right\|\right)_k$ and $\left(\left\|x^k - x^*\right\|\right)_k$ (with the iteration number $k$ as $x$ -axis) in a logarithmic scale and compare the performance of the gradient method using the different step size strategies mentioned in part a). Here, $x^*$ denotes the limit point of the sequence generated by the gradient method. Which type of convergences can be observed?

c) Let us define the set(this is a rectangle).

$$\mathcal{X}^0 := \left\{x \in \mathbb{R}^2 : x_1 \in \{-10, 10\}, x_2 \in [-2, 2]\right\} \cup \left\{x \in \mathbb{R}^2 : x_1 \in [-10, 10], x_2 \in \{-2, 2\}\right\}$$

Run the methods:

- Gradient descent method with backtracking and $(s, \sigma, \gamma) = (1, 0.5, 0.1)$
- Gradient method with diminishing step sizes $\alpha_k = 0.01/\log(k + 2)$
- Gradient method with exact line search and maxit $= 100$, tol $= 10^{-6}$, a $= 2$

again with $p$ different initial points selected from the set $\mathcal{X}^0$. The initial points should uniformly cover the different parts of the set $\mathcal{X}^0$ and you can use the tolerance tol $= 10^{-5}$ and $p \in [10, 20] \cap \mathbb{N}$. For each algorithm create a single figure that contains all of the solution paths generated for the different initial points. The initial points and limit points should be clearly visible. Add a contour plot of the function $f$ in the background of each figure.

**Hint**: You might need to adjust the diminishing step sizes to still guarantee convergence.

## Solution

### Subproblem (a)

∘ Part 1: The calculation of stationary points.
The gradient of function $f$ is as follow:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 2f_1(x) + 2f_2(x) \\ 2f_1(x)(2x_2 - 3x_2^2 - 2) + 2f_2(x)(2x_2 - 3) \end{pmatrix} \tag{7}$$

The Hessian of function $f$ is as follow:

$$\nabla^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} \end{pmatrix} = \begin{pmatrix} 4 & 8x_2 - 6x_2^2 - 10 \\ 8x_2 - 6x_2^2 - 10 & 2f_1(x)(-6x_2 + 2) + 2(2x_2 - 3x_2^2 - 2)^2 + 4f_2(x) + 2(2x_2 - 3)^2 \end{pmatrix} \tag{8}$$

By solving the equation $\nabla f(x) = 0$, we derive the stationary points as follow:

$$A(-7,-1) \quad B(-3,0) \quad C(-1,1) \quad D\left(-\frac{2}{9}\left(15+4\sqrt{3}\right), -\frac{1}{\sqrt{3}}\right) \quad E\left(\frac{2}{9}\left(4\sqrt{3}-15\right), \frac{1}{\sqrt{3}}\right)$$

By calculating the Hessian matrix for point $A, B, C, D, E$, we can find that 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 $f(x) \geq 0$, 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 2: The gradient method
By using different step size strategies (Backtracking, Exact line search and Diminishing step size method), we find that with the initial point (0, 0), all the method converges to the point (-1, 1) approximately. The number of iterations with different step size strategies are 162 (Backtracking), 11993 (Diminishing step size), and 7 (Exact line search). As we can see from the result, "Exact line search" is the fastest method, and "Diminishing step size" is the lowest method for this certain problem. The python code to solve this problem is as follow:

```
import numpy as np

def f1(x):
  return 3 + x[0] + ((1 - x[1]) * x[1] - 2) * x[1]
def f2(x):
  return 3 + x[0] + (x[1] - 3) * x[1]
def f(x):
  return f1(x) * f1(x) + f2(x) *f2(x)
def df(x):
  grad = np.zeros(2).reshape(2,1)
  grad[0] = 2 * f1(x) + 2 * f2(x)
  grad[1] = 2 * f1(x) * (2*x[1] - 3*(x[1]**2) - 2) + 2 * f2(x) * (2*x[1] - 3)
  return grad
def norm(x):
  return np.sqrt(x[0]**2 + x[1]**2)

#backtracking method
s = 1
sigma = 0.5
gamma = 0.1
alpha = s
tol = 1e-5
```

```
initial = np.array([0, 0]).reshape(2, 1)
xk = initial
gradient = df(xk)
num_iteration = 0

while norm(gradient) > tol:
  alphak = s
  dk = -df(xk)
  while True:
    if f(xk + alphak*dk) - f(xk) <= gamma * alphak * np.dot(df(xk).T, dk):
      break
    alphak = alphak * sigma
  xk = xk + alphak * dk
  gradient = df(xk)
  num_iteration = num_iteration + 1
print('--------backtracking---------')
print('xk:\n', xk)
print('number of iterations:', num_iteration)
print()

#diminishing step size
def alpha_k(k):
  return 0.01/(np.log(k+2))

num_iteration = 0
k = 1
xk = initial
gradient = df(xk)

while norm(gradient) > tol:
  dk = -df(xk)
  alphak = alpha_k(k)
  xk = xk + alphak * dk
  gradient = df(xk)
  num_iteration = num_iteration + 1
  k = k + 1
print('--------diminishing step size---------')
print('xk:\n', xk)
print('number of iterations:', num_iteration)
print()

# exact line search
tol_golden = 1e-6
maxit = 100
a = 2
num_iteration = 0
xk = initial
gradient = df(xk)

while norm(gradient) > tol:
  dk = -df(xk)
  alphal = 0
  alphar = a
  num_iteration_golden = 0
  last_update = 0 #0:left; 1:right
  fi = (3 - np.sqrt(5)) / 2
  while (alphar - alphal) >= tol_golden and num_iteration_golden < 100:
    if num_iteration_golden == 0:
      new_alphal = fi * alphar + (1 - fi) * alphal
      new_alphar = (1 - fi) * alphar + fi * alphal
    elif last_update == 0:
      new_alphal = new_alphar
```

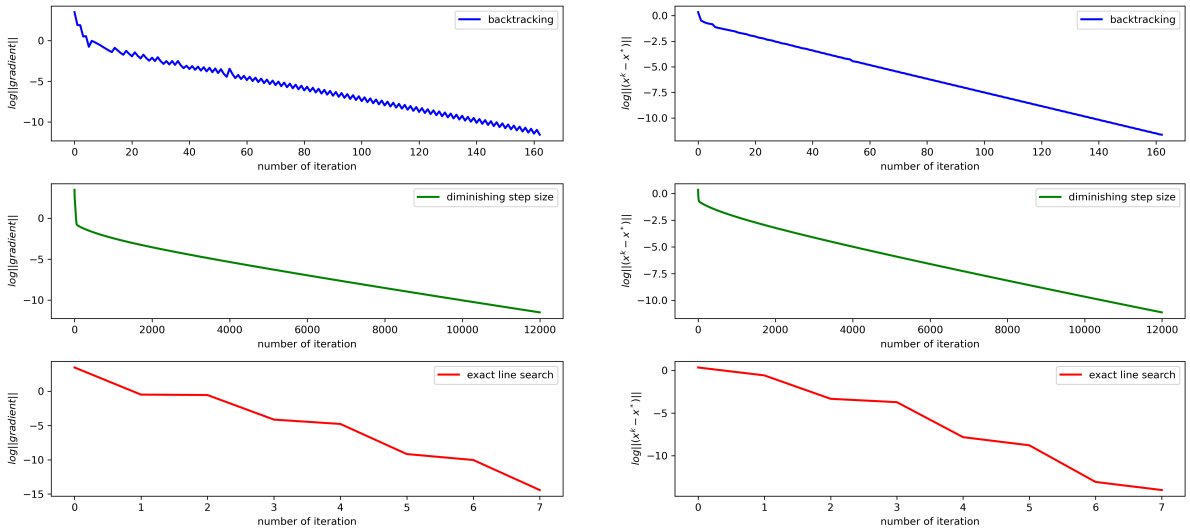
```
      new_alphar = (1 - fi) * alphar + fi * alphal
    else:
      new_alphar = new_alphal
      new_alphal = fi * alphar + (1 - fi) * alphal

    f_alphal = f(xk + new_alphal*dk)
    f_alphar = f(xk + new_alphar*dk)

    if f_alphal < f_alphar:
      alphar = new_alphar
      last_update = 1
    else:
      alphal = new_alphal
      last_update = 0
    num_iteration_golden = num_iteration_golden + 1
  alphak = (alphal + alphar) / 2
  xk = xk + alphak * dk
  gradient = df(xk)
  num_iteration = num_iteration + 1
print('--------eaxct line search---------')
print('xk:\n', xk)
print('number of iterations:', num_iteration)
print()
```

**Subproblem (b)**

From problem (a), we know the limit points of all the three sequences generated by gradient method with different step size method are (-1, 1). So $x^* := (-1, 1)$. The figure of $\left(\left\|\nabla f\left(x^k\right)\right\|\right)_k$ and $\left(\left\|x^k - x^*\right\|\right)_k$ can be seen in Figure 1.




(a) $\log \|\nabla f(x^k)\|$ VS number of iteration (b) $\log \|x^k - x^*\|$ VS number of iteration

**Figure 1 The plot of iteration process**

As we can see from the figure, the convergence type is linear convergence.
The python code to solve this problem is showing below.

```
import numpy as np
import matplotlib.pyplot as plt

def f1(x):
    return 3 + x[0] + ((1 - x[1]) * x[1] - 2) * x[1]
def f2(x):
    return 3 + x[0] + (x[1] - 3) * x[1]
def f(x):
    return f1(x) * f1(x) + f2(x) *f2(x)
def df(x):
    grad = np.zeros(2).reshape(2,1)
    grad[0] = 2 * f1(x) + 2 * f2(x)
    grad[1] = 2 * f1(x) * (2*x[1] - 3*(x[1]**2) - 2) + 2 * f2(x) * (2*x[1] - 3)
    return grad
def norm(x):
    return np.sqrt(x[0]**2 + x[1]**2)
color_list = ['blue', 'green', 'red', 'cyan']
def plot_gradient(y, method, subfig_num):
    plt.figure(1, figsize=(8,10))
    plt.subplot(3,1,subfig_num)
    n = y.size
    x = np.arange(n)
    y = np.log(y)
    plt.plot(x, y, label=method, color = color_list[subfig_num-1], linewidth=2)
    plt.legend()
    plt.xlabel('number of iteration')
    plt.ylabel('$log||gradient||$')
    plt.tight_layout()
    plt.savefig('gradient', dpi=300)

def plot_xk(y, method, subfig_num):
    plt.figure(2, figsize=(8,10))
    plt.subplot(3,1,subfig_num)
    n = y.size
    x = np.arange(n)
    y = np.log(y)
    plt.plot(x, y, label = method, color = color_list[subfig_num-1], linewidth=2)
    plt.legend()
    plt.xlabel('number of iteration')
    plt.ylabel('$log||(x^k-x^*)||$')
    plt.tight_layout()
    plt.savefig('xk', dpi=300)

#backtracking method
x_star = np.array([-1, 1]).reshape(2, 1)
s = 1
sigma = 0.5
gamma = 0.1
alpha = s
tol = 1e-5
gradient_list = []
xk_list = []

initial = np.array([0, 0]).reshape(2, 1)
xk = initial
gradient = df(xk)
num_iteration = 0
gradient_list.append(norm(gradient))
xk_list.append(norm(xk-x_star))

while norm(gradient) > tol:
```

```
    alphak = s
    dk = -df(xk)
    while True:
        if f(xk + alphak*dk) - f(xk) <= gamma * alphak * np.dot(df(xk).T, dk):
            break
        alphak = alphak * sigma
    xk = xk + alphak * dk
    xk_list.append(norm(xk-x_star))
    gradient = df(xk)
    gradient_list.append(norm(gradient))
    num_iteration = num_iteration + 1

xk_list = np.array(xk_list)
gradient_list = np.array(gradient_list)
subfig_num = 1
method = 'backtracking'
plot_gradient(gradient_list, method, subfig_num)
plot_xk(xk_list, method, subfig_num)

print('--------backtracking---------')
print('xk:\n', xk)
print('number of iterations:', num_iteration)
print()

#diminishing step size
def alpha_k(k):
    return 0.01/(np.log(k+2))

num_iteration = 0
k = 1
xk = initial
gradient = df(xk)
gradient_list = []
xk_list = []
gradient_list.append(norm(gradient))
xk_list.append(norm(xk-x_star))

while norm(gradient) > tol:
    dk = -df(xk)
    alphak = alpha_k(k)
    xk = xk + alphak * dk
    xk_list.append(norm(xk-x_star))
    gradient = df(xk)
    gradient_list.append(norm(gradient))
    num_iteration = num_iteration + 1
    k = k + 1
xk_list = np.array(xk_list)
gradient_list = np.array(gradient_list)
subfig_num = 2
method = 'diminishing step size'
plot_gradient(gradient_list, method, subfig_num)
plot_xk(xk_list, method, subfig_num)
print('--------diminishing step size---------')
print('xk:\n', xk)
print('number of iterations:', num_iteration)
print()

# exact line search
tol_golden = 1e-6
maxit = 100
a = 2
num_iteration = 0
```

```
xk = initial
gradient = df(xk)
gradient_list = []
xk_list = []
gradient_list.append(norm(gradient))
xk_list.append(norm(xk-x_star))

while norm(gradient) > tol:
    dk = -df(xk)
    alphal = 0
    alphar = a
    num_iteration_golden = 0
    last_update = 0 #0:left; 1:right
    fi = (3 - np.sqrt(5)) / 2
    while (alphar - alphal) >= tol_golden and num_iteration_golden < 100:
        if num_iteration_golden == 0:
            new_alphal = fi * alphar + (1 - fi) * alphal
            new_alphar = (1 - fi) * alphar + fi * alphal
        elif last_update == 0:
            new_alphal = new_alphar
            new_alphar = (1 - fi) * alphar + fi * alphal
        else:
            new_alphar = new_alphal
            new_alphal = fi * alphar + (1 - fi) * alphal

        f_alphal = f(xk + new_alphal*dk)
        f_alphar = f(xk + new_alphar*dk)

        if f_alphal < f_alphar:
            alphar = new_alphar
            last_update = 1
        else:
            alphal = new_alphal
            last_update = 0
        num_iteration_golden = num_iteration_golden + 1
    alphak = (alphal + alphar) / 2
    xk = xk + alphak * dk
    xk_list.append(norm(xk-x_star))
    gradient = df(xk)
    gradient_list.append(norm(gradient))
    num_iteration = num_iteration + 1
xk_list = np.array(xk_list)
gradient_list = np.array(gradient_list)
subfig_num = 3
method = 'exact line search'
plot_gradient(gradient_list, method, subfig_num)
plot_xk(xk_list, method, subfig_num)

print('--------eaxct line search---------')
print('xk:\n', xk)
print('number of iterations:', num_iteration)
print()
```

**Subproblem (c)**

We choose $p$ =12 different initial points as (-10, ±2), (-6, ±2), (-2, ±2), (2, ±2), (6, ±2), (10, ±2). In diminishing step method, we adjust the diminishing step sizes as $\alpha_k = 0.01/\log(k+15)$. The figures which contains the paths is showing as Figure 2, Figure 3 and Figure 4 for backtracking, diminishing step size and exact line search respectively. Besides, the python code to solve this problem is showing below.

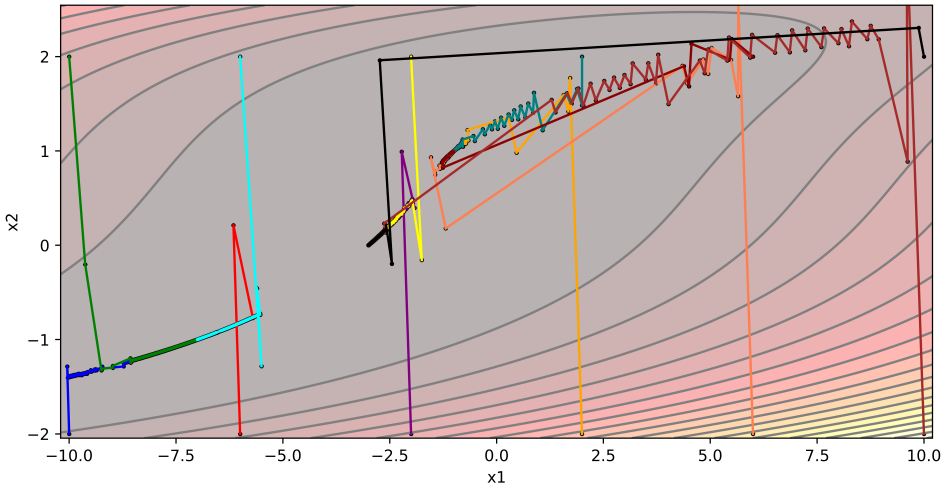



**Figure 2 Paths of backtracking with different initial points**

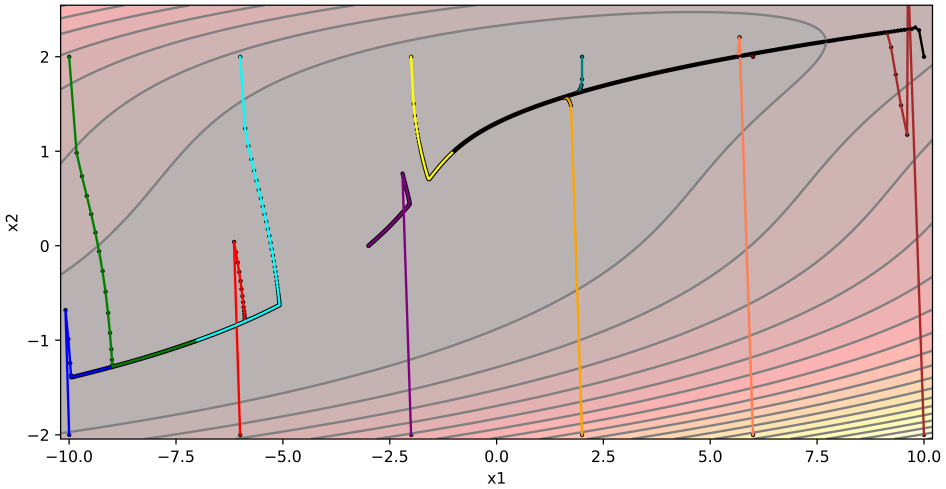



**Figure 3 Paths of diminishing step size with different initial points**

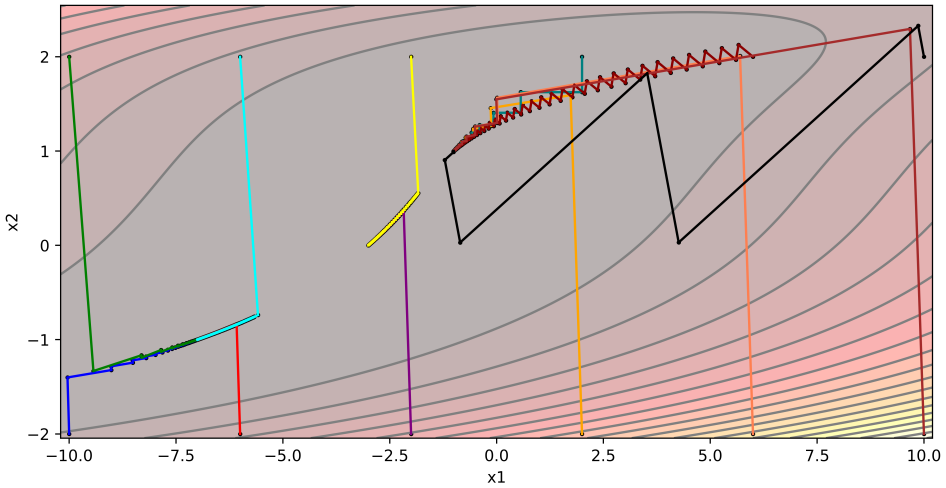



**Figure 4 Paths of exact line search with different initial points**

```
# backtracking method
import numpy as np
import matplotlib.pyplot as plt

def f1(x):
    return 3 + x[0] + ((1 - x[1]) * x[1] - 2) * x[1]
def f2(x):
    return 3 + x[0] + (x[1] - 3) * x[1]
def f(x):
    return f1(x) * f1(x) + f2(x) *f2(x)
def df(x):
    grad = np.zeros(2).reshape(2,1)
    grad[0] = 2 * f1(x) + 2 * f2(x)
    grad[1] = 2 * f1(x) * (2*x[1] - 3*(x[1]**2) - 2) + 2 * f2(x) * (2*x[1] - 3)
    return grad
def norm(x):
    return np.sqrt(x[0]**2 + x[1]**2)

color_list = ['blue', 'green', 'red', 'cyan', 'purple', 'yellow', 'orange', 'teal',
              'coral', 'darkred', 'brown', 'black']
def plot_contour():
    X = np.arange(-10.2,10.21,0.05)
    Y = np.arange(-2.045,2.55,0.01125)
    X,Y = np.meshgrid(X,Y)
    Z = np.zeros((X.shape[0], X.shape[1]))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            x = []
            x.append(X[i][j])
            x.append(Y[i][j])
            x = np.array(x)
            Z[i][j] = f(x)
    plt.contourf(X, Y, Z, 20, alpha=0.3, cmap=plt.cm.hot)
    plt.contour(X, Y, Z, 20, colors='grey')
def plot_line(xk_list, subfig_num):
    x = []
    y = []
    for i in range(xk_list.shape[0]):
        x.append(xk_list[i][0][0])
        y.append(xk_list[i][1][0])
    plt.plot(x,y, color = color_list[subfig_num-1], linewidth=1.5)
    plt.scatter(x, y, s=3, color='black')

def gradient_method(initial, subfig_num):
    s = 1
    sigma = 0.5
    gamma = 0.1
    tol = 1e-5
    xk_list = []

    xk = initial
    gradient = df(xk)
    num_iteration = 0
    xk_list.append(xk)

    while norm(gradient) > tol:
        alphak = s
        dk = -df(xk)
        while True:
            if f(xk + alphak*dk) - f(xk) <= gamma * alphak * np.dot(df(xk).T, dk):
                break
```

```
            alphak = alphak * sigma
        xk = xk + alphak * dk
        xk_list.append(xk)
        gradient = df(xk)
        num_iteration = num_iteration + 1

    xk_list = np.array(xk_list)
    plot_line(xk_list, subfig_num)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.xlim(-10.2, 10.2)
    plt.ylim(-2.045, 2.545)

x1 = np.arange(-10, 11, 4)
x2 = np.arange(-2, 3, 4)

plt.figure(1, figsize=(10, 5))
plot_contour()
subfig_num = 1
for i in range(6):
    for j in range(2):
        initial = np.zeros(2).reshape(2,1)
        initial[0] = x1[i]
        initial[1] = x2[j]
        gradient_method(initial, subfig_num)
        subfig_num = subfig_num + 1
plt.savefig('backtracking', dpi=700)
```

```
# diminishing step size
import numpy as np
import matplotlib.pyplot as plt

def f1(x):
    return 3 + x[0] + ((1 - x[1]) * x[1] - 2) * x[1]
def f2(x):
    return 3 + x[0] + (x[1] - 3) * x[1]
def f(x):
    return f1(x) * f1(x) + f2(x) *f2(x)
def df(x):
    grad = np.zeros(2).reshape(2,1)
    grad[0] = 2 * f1(x) + 2 * f2(x)
    grad[1] = 2 * f1(x) * (2*x[1] - 3*(x[1]**2) - 2) + 2 * f2(x) * (2*x[1] - 3)
    return grad
def norm(x):
    return np.sqrt(x[0]**2 + x[1]**2)
def alpha_k(k):
    return 0.01/(np.log(k+15))

color_list = ['blue', 'green', 'red', 'cyan', 'purple', 'yellow', 'orange', 'teal',
              'coral', 'darkred', 'brown', 'black']
def plot_contour():
    X = np.arange(-10.2,10.21,0.05)
    Y = np.arange(-2.045,2.55,0.01125)
    X,Y = np.meshgrid(X,Y)
    Z = np.zeros((X.shape[0], X.shape[1]))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            x = []
            x.append(X[i][j])
            x.append(Y[i][j])
            x = np.array(x)
            Z[i][j] = f(x)
```

```
    plt.contourf(X, Y, Z, 20, alpha=0.3, cmap=plt.cm.hot)
    plt.contour(X, Y, Z, 20, colors='grey')
def plot_line(xk_list, subfig_num):
    x = []
    y = []
    for i in range(xk_list.shape[0]):
        x.append(xk_list[i][0][0])
        y.append(xk_list[i][1][0])
    plt.plot(x,y, color = color_list[subfig_num-1], linewidth=1.5)
    plt.scatter(x, y, s=3, color='black')

def gradient_method(initial, subfig_num):
    tol = 1e-5
    num_iteration = 0
    k = 1
    xk = initial
    gradient = df(xk)
    xk_list = []
    xk_list.append(xk)

    while norm(gradient) > tol:
        dk = -df(xk)
        alphak = alpha_k(k)
        xk = xk + alphak * dk
        xk_list.append(xk)
        gradient = df(xk)
        num_iteration = num_iteration + 1
        k = k + 1

    xk_list = np.array(xk_list)
    plot_line(xk_list, subfig_num)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.xlim(-10.2, 10.2)
    plt.ylim(-2.045, 2.545)

x1 = np.arange(-10, 11, 4)
x2 = np.arange(-2, 3, 4)

plt.figure(1, figsize=(10, 5))
plot_contour()
subfig_num = 1

for i in range(6):
    for j in range(2):
        initial = np.zeros(2).reshape(2,1)
        initial[0] = x1[i]
        initial[1] = x2[j]
        gradient_method(initial, subfig_num)
        subfig_num = subfig_num + 1
plt.savefig('diminishing', dpi=700)
```

```
# exact line search
import numpy as np
import matplotlib.pyplot as plt

def f1(x):
    return 3 + x[0] + ((1 - x[1]) * x[1] - 2) * x[1]
def f2(x):
    return 3 + x[0] + (x[1] - 3) * x[1]
def f(x):
    return f1(x) * f1(x) + f2(x) *f2(x)
```

Assignment A3.3 continued on next page... 

```
def df(x):
    grad = np.zeros(2).reshape(2,1)
    grad[0] = 2 * f1(x) + 2 * f2(x)
    grad[1] = 2 * f1(x) * (2*x[1] - 3*(x[1]**2) - 2) + 2 * f2(x) * (2*x[1] - 3)
    return grad
def norm(x):
    return np.sqrt(x[0]**2 + x[1]**2)

color_list = ['blue', 'green', 'red', 'cyan', 'purple', 'yellow', 'orange', 'teal',
              'coral', 'darkred', 'brown', 'black']
def plot_contour():
    X = np.arange(-10.2,10.21,0.05)
    Y = np.arange(-2.045,2.55,0.01125)
    X,Y = np.meshgrid(X,Y)
    Z = np.zeros((X.shape[0], X.shape[1]))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            x = []
            x.append(X[i][j])
            x.append(Y[i][j])
            x = np.array(x)
            Z[i][j] = f(x)
    plt.contourf(X, Y, Z, 20, alpha=0.3, cmap=plt.cm.hot)
    plt.contour(X, Y, Z, 20, colors='grey')
def plot_line(xk_list, subfig_num):
    x = []
    y = []
    for i in range(xk_list.shape[0]):
        x.append(xk_list[i][0][0])
        y.append(xk_list[i][1][0])
    plt.plot(x,y, color = color_list[subfig_num-1], linewidth=1.5)
    plt.scatter(x, y, s=3, color='black')

def gradient_method(initial, subfig_num):
    tol = 1e-5
    xk = initial
    gradient = df(xk)
    num_iteration = 0
    tol_golden = 1e-6
    maxit = 100
    a = 2
    xk_list = []
    xk_list.append(xk)

    while norm(gradient) > tol:
        dk = -df(xk)
        alphal = 0
        alphar = a
        num_iteration_golden = 0
        last_update = 0 #0:left; 1:right
        fi = (3 - np.sqrt(5)) / 2
        while (alphar - alphal) >= tol_golden and num_iteration_golden < maxit:
            if num_iteration_golden == 0:
                new_alphal = fi * alphar + (1 - fi) * alphal
                new_alphar = (1 - fi) * alphar + fi * alphal
            elif last_update == 0:
                new_alphal = new_alphar
                new_alphar = (1 - fi) * alphar + fi * alphal
            else:
                new_alphar = new_alphal
                new_alphal = fi * alphar + (1 - fi) * alphal
```

```
            f_alphal = f(xk + new_alphal*dk)
            f_alphar = f(xk + new_alphar*dk)

            if f_alphal < f_alphar:
                alphar = new_alphar
                last_update = 1
            else:
                alphal = new_alphal
                last_update = 0
            num_iteration_golden = num_iteration_golden + 1
        alphak = (alphal + alphar) / 2
        xk = xk + alphak * dk
        xk_list.append(xk)
        gradient = df(xk)
        num_iteration = num_iteration + 1

    xk_list = np.array(xk_list)
    plot_line(xk_list, subfig_num)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.xlim(-10.2, 10.2)
    plt.ylim(-2.045, 2.545)

x1 = np.arange(-10, 11, 4)
x2 = np.arange(-2, 3, 4)

plt.figure(1, figsize=(10, 5))
plot_contour()
subfig_num = 1
for i in range(6):
    for j in range(2):
        initial = np.zeros(2).reshape(2,1)
        initial[0] = x1[i]
        initial[1] = x2[j]
        gradient_method(initial, subfig_num)
        subfig_num = subfig_num + 1
plt.savefig('exact', dpi=700)
```