1 Steepest Descent Method in 2D

To solve the following minimization problem,

$$\min_{x} f(x) \tag{1}$$

in this section, we study steepest descent method(or gradient descent method) expressed as for given $x_0 \in \mathbf{R}^2$, do iteration for $k = 0, \cdots, M-1$

$$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x^{(k)}) \tag{2}$$

```python
import numpy as np
def steepest_descent_2d(func, gradx, grady, x0, MaxIter=10, learning_rate=0.25):
    for i in range(MaxIter):
        grad = np.array([gradx(*x0), grady(*x0)])
        x1 = x0 - learning_rate * grad
        x0 = x1
    return x0
```

1. Start with an initial $x_0 \in \mathbf{R}^2$. For example,

   ```python
   import numpy as np
   x0 = np.array([-2.0, -2.0])
   ```

2. Do $k = 0, 1, \cdots,$ `MaxIter-1`

   ```python
   for i in range(MaxIter):
   ```

   (a) Calculate its gradient, $\nabla f(x^{(k)})$

   ```python
   grad = np.array([gradx(*x0), grady(*x0)])
   ```

   - `gradx()` : function for $\frac{\partial f}{\partial x}$
   - `grady()` : function for $\frac{\partial f}{\partial y}$
   - `x0` : current position $x^{(k)}$
   - `grad` : gradient vector at curret position $\nabla f(x^{(k)}) \in \mathbf{R}^2$

   (b) Calculate next position $x^{(k+1)}$ with learning rate α as follows

   $$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x^{(k)}) \tag{3}$$

   ```python
   x1 = x0 - learning_rate * grad
   ```

   - `x1` : next position $x^{(k+1)}$
   - `learning_rate` : α

   (c) Update old one to new one

   ```python
   x0 = x1
   ```

Example 1.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{4}$$

1. Define $f(x, y) = 3(x-2)^2 + (y-2)^2$

   ```python
   f = lambda x,y : 3 * (x - 2)**2 + (y - 2)**2
   ```

2. Define $\frac{\partial f}{\partial x} = 6(x-2)$

   ```python
   grad_x = lambda x,y : 6 * (x - 2)
   ```

3. Define $\frac{\partial f}{\partial y} = 2(y-2)$

   ```python
   grad_y = lambda x,y : 2 * (y - 2)
   ```

4. Tune parameters such as `x0, learning_rate, MaxIter`

5. Run steepest descent scheme!

```python
import numpy as np
def steepest_descent_2d(func, gradx, grady, x0, MaxIter=10, learning_rate=0.25):
    for i in range(MaxIter):
        grad = np.array([gradx(*x0), grady(*x0)])
        x1 = x0 - learning_rate * grad
        x0 = x1
    return x0

# Define functions for the problem
f = lambda x,y : 3 * (x - 2)**2 + (y - 2)**2
grad_x = lambda x,y : 6 * (x - 2)
grad_y = lambda x,y : 2 * (y - 2)

# Tune parameters
x0 = np.array([-2.0, -2.0])
learning_rate = 0.1
MaxIter = 100
xopt = steepest_descent_2d(f, grad_x, grad_y, x0,
                MaxIter=MaxIter, learning_rate=learning_rate)
# Result will be [ 2.  2.]
print(xopt)
```

2 Newton method in 2D

To solve the following minimization problem,

$$\min_x f(x) \tag{5}$$

in this section, we study Newton method expressed as for given $x_0 \in \mathbf{R}^2$, do iteration for $k = 0, \cdots, M-1$

$$x^{(k+1)} = x^{(k)} - \left[\nabla^2 f(x^{(k)})\right]^{-1} \nabla f(x^{(k)}) \tag{6}$$

where

$$\nabla^2 f(x) = \begin{bmatrix} \partial^2_{xx} f & \partial^2_{yx} f \\ \partial^2_{xy} f & \partial^2_{yy} f \end{bmatrix}. \tag{7}$$

```
import numpy as np
def newton_descent_2d(func, gradx, grady, hessian, x0, MaxIter=10, learning_rate=1):
    for i in range(MaxIter):
        grad = np.array([gradx(*x0), grady(*x0)])
        hess = hessian(*x0)
        delx = np.linalg.solve(hess, grad)
        x1 = x0 - learning_rate * delx
        x0 = x1
    return x0
```

1. Start with an initial $x_0 \in \mathbf{R}^2$. For example,

```
import numpy as np
x0 = np.array([-2.0, -2.0])
```

2. Do $k = 0, 1, \cdots,$ `MaxIter-1`

```
for i in range(MaxIter):
```

   (a) Calculate its gradient, $\nabla f(x^{(k)})$, of $f(x^{(k)})$ at $x^{(k)}$

   ```
   grad = np.array([gradx(*x0), grady(*x0)])
   ```

   - `gradx()` : function for $\frac{\partial f}{\partial x}$
   - `grady()` : function for $\frac{\partial f}{\partial y}$
   - `x0` : current position $x^{(k)}$
   - `grad` : gradient vector at curret position $\nabla f(x^{(k)}) \in \mathbf{R}^2$

   (b) Calculate its Hessian, $\nabla^2 f(x^{(k)})$

   ```
   hess = hessian(*x0)
   ```

   - `hessian()` : function for $\nabla^2 f(x)$
   - `hess` : Hessian matrix $\nabla^2 f(x^{(k)}) \in \mathbf{R}^{2\times 2}$

   (c) Solve linear system : $\left[\nabla^2 f(x^{(k)})\right] \Delta x^{(k)} = \nabla f(x^{(k)})$

   ```
   delx = np.linalg.solve(hess, grad)
   ```

   - `np.linalg.solve(A,b)` : method for solving linear system, $Ax = b$
   - `delx` : $\Delta x^{(k)}$

   (d) Calculate next position $x^{(k+1)}$ with learning rate α as follows

   $$x^{(k+1)} = x^{(k)} - \alpha \left[\nabla^2 f(x^{(k)})\right]^{-1} \nabla f(x^{(k)}) \tag{8}$$

   $$= x^{(k)} - \alpha \Delta x^{(k)} \tag{9}$$

   ```
   x1 = x0 - learning_rate * delx
   ```

   - `x1` : next position $x^{(k+1)}$
   - `learning_rate` : α

(e) Update old one to new one

```
x0 = x1
```

Example 2.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{10}$$

1. Define $f(x,y) = 3(x-2)^2 + (y-2)^2$

```
f = lambda x,y : 3 * (x - 2)**2 + (y - 2)**2
```

2. Define $\frac{\partial f}{\partial x} = 6(x-2)$

```
grad_x = lambda x,y : 6 * (x - 2)
```

3. Define $\frac{\partial f}{\partial y} = 2(y-2)$

```
grad_y = lambda x,y : 2 * (y - 2)
```

4. Define $\nabla^2 f$

$$\begin{aligned}\nabla^2 f(x) &= \begin{bmatrix} \partial^2_{xx} f & \partial^2_{yx} f \\ \partial^2_{xy} f & \partial^2_{yy} f \end{bmatrix} && (11) \\ &= \begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix} && (12)\end{aligned}$$

```
hessian = lambda x,y : np.array([[6., 0.],[0., 2.]])
```

5. Tune parameters such as `x0, learning_rate, MaxIter`

6. Run Newton method!

```python
import numpy as np
def newton_descent_2d(func, gradx, grady, hessian, x0, MaxIter=10, learning_rate=1):
    for i in range(MaxIter):
        grad = np.array([gradx(*x0), grady(*x0)])
        hess = hessian(*x0)
        delx = np.linalg.solve(hess, grad)
        x1 = x0 - learning_rate * delx
        x0 = x1
    return x0

# Define functions for the problem
f = lambda x,y : 3 * (x - 2)**2 + (y - 2)**2
grad_x = lambda x,y : 6 * (x - 2)
grad_y = lambda x,y : 2 * (y - 2)
hessian = lambda x,y : np.array([[6., 0.],[0., 2.]])

# Tune parameters(Use default values for MaxIter, learning_rate)
x0 = np.array([-2.0, -2.0])
xopt = newton_descent_2d(f, grad_x, grad_y, hessian, x0)

# Result will be [ 2.  2.]
print(xopt)
```

3 BFGS Method in 2D

To solve the following minimization problem,

$$\min_x f(x) \tag{13}$$

in this section, we study BFGS method expressed as for given $x_0 \in \mathbf{R}^2$ and $B_0 \in \mathbf{R}^{2\times 2}$, do iteration for $k = 0, \cdots, M-1$

$$p_k = -B_k^{-1}\nabla f(x_k) \tag{14}$$
$$\Delta x_k = \alpha p_k \tag{15}$$
$$x_{k+1} = x_k + \Delta x_k \tag{16}$$
$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k) \tag{17}$$
$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T \Delta x_k} - \frac{B_k \Delta x_k \Delta x_k^T B_k}{\Delta x_k^T B_k \Delta x_k} \tag{18}$$

```python
import numpy as np
def bfgs_method_2d(func, gradx, grady, x0, MaxIter=10, learning_rate=1):
    B0 = np.eye(len(x0))
    for i in range(MaxIter):
        grad = np.array([gradx(*x0), grady(*x0)])
        p0 = -np.linalg.solve(B0, grad)
        delx = learning_rate * p0
        x1 = x0 + delx
        y0 = (np.array([gradx(*x1), grady(*x1)]) - grad).reshape(-1,1)
        B1 = B0 + np.dot(y0, y0.T) / np.dot(y0.T, delx) \
                - np.dot(np.dot(B0, delx).reshape(-1,1), np.dot(delx, B0).reshape(-1,1).T) \
                / np.dot(np.dot(B0, delx), delx)
        x0 = x1
        B0 = B1
    return x0
```

1. Start with initial x_0 and B_0.

```python
import numpy as np
x0 = np.array([-2.0, -2.0])
B0 = np.eye(len(x0))
```

2. Do $k = 0, 1, \cdots,$ `MaxIter-1`,

```python
for i in range(MaxIter):
```

   (a) Calculate its gradient, $\nabla f(x_k)$

   ```python
   grad = np.array([gradx(*x0), grady(*x0)])
   ```

   (b) Solve linear system

   $$p_k = -B_k \nabla f(x_k) \tag{19}$$

   ```python
   p0 = -np.linalg.solve(B0, grad)
   ```

   - `grad` : gradient of f at x_k
   - `B0` : approximation of hessian matrix $\nabla^2 f(x_k)$

   (c) Set search direction, Δx_k, and update next position, x_{k+1}

   $$\Delta x_k = \alpha p_k \tag{20}$$
   $$x_{k+1} = x_k + \Delta x_k \tag{21}$$

   ```python
   delx = learning_rate * p0
   x1 = x0 + delx
   ```

   - `delx` : update size

- `x1` : next position

(d) Calculate y_k

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k) \tag{22}$$

```
y0 = (np.array([gradx(*x1), grady(*x1)]) - grad).reshape(-1,1)
```

(e) Update old ones to new ones.

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T \Delta x_k} - \frac{B_k \Delta x_k \Delta x_k^T B_k}{\Delta x_k^T B_k \Delta x_k} \tag{23}$$

```
B1 = B0 + np.dot(y0, y0.T) / np.dot(y0.T, delx)
        - np.dot(np.dot(B0, delx).reshape(-1,1), np.dot(delx, B0).reshape(-1,1).T)
        / np.dot(np.dot(B0, delx), delx)
```

- `B1` : next approximation of hessian matrix $\nabla^2 f(x_{k+1})$

(f) Update x_{k+1} and B_{k+1}

```
x0 = x1
B0 = B1
```

Example 3.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{24}$$

1. Define $f(x, y) = 3(x-2)^2 + (y-2)^2$

```
f = lambda x,y : 3 * (x - 2)**2 + (y - 2)**2
```

2. Define $\frac{\partial f}{\partial x} = 6(x-2)$

```
grad_x = lambda x,y : 6 * (x - 2)
```

3. Define $\frac{\partial f}{\partial y} = 2(y-2)$

```
grad_y = lambda x,y : 2 * (y - 2)
```

4. Tune parameters such as `x0, learning_rate, MaxIter`

5. Run BFGS method!

```
import numpy as np
def bfgs_method_2d(func, gradx, grady, x0, MaxIter=10, learning_rate=1):
    B0 = np.eye(len(x0))
    for i in range(MaxIter):
        grad = np.array([gradx(*x0), grady(*x0)])
        p0 = -np.linalg.solve(B0, grad)
        delx = learning_rate * p0
        x1 = x0 + delx
        y0 = (np.array([gradx(*x1), grady(*x1)]) - grad).reshape(-1,1)
        B1 = B0 + np.dot(y0, y0.T) / np.dot(y0.T, delx) \
                - np.dot(np.dot(B0, delx).reshape(-1,1), np.dot(delx, B0).reshape(-1,1).T) \
                / np.dot(np.dot(B0, delx), delx)
        x0 = x1
        B0 = B1
    return x0

# Define functions for the problem
f = lambda x,y : 3 * (x - 2)**2 + (y - 2)**2
grad_x = lambda x,y : 6 * (x - 2)
grad_y = lambda x,y : 2 * (y - 2)
hessian = lambda x,y : np.array([[6., 0.],[0., 2.]])

```

```python
# Tune parameters(Use default values for MaxIter, learning_rate)
x0 = np.array([-2.0, -2.0])
xopt = bfgs_method_2d(f, grad_x, grad_y, x0, MaxIter=6)

# Result will be [ 2.  2.]
print(xopt)
```