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

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

```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
```

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, $\nabla f(x_k)$, of $f(x_k)$ at 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 its Hessian, $\nabla^2 f(x_k)$

```python
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 : $\left[\nabla^2 f(x_k)\right] \Delta x_k = \nabla f(x_k)$

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

   - `np.linalg.solve(A,b)` : method for solving linear system, $Ax = b$
   - `delx` : Δx_k

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

$$\begin{aligned} x_{k+1} &= x_k - \alpha \left[\nabla^2 f(x_k)\right]^{-1} \nabla f(x_k) && (8) \\ &= x_k - \alpha \Delta x_k && (9) \end{aligned}$$

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

   - `x1` : next position x_{k+1}
   - `learning_rate` : α

   (e) Update old one to new one

```python
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 $\nabla^2 f$

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

$$= \begin{bmatrix} 6 & 0 \\ 0 & 2 \end{bmatrix} \tag{12}$$

```
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 $k = 0, \cdots, M-1$

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

```
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.

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

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

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

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

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

   (b) Solve linear system

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

   ```
   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}

   $$\Delta x_k = \alpha p_k \tag{20}$$

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

   ```
   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.]])

# 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)
```

4 Nesterov Momentum in 2D

To solve the following minimization problem,

$$\min_{x} f(x) \tag{25}$$

in this section, we study Nesterov Momentum method (see [Nesterov, 1983]) expressed as for given $x_0 \in \mathbf{R}^2$, do iteration for $k = 0, \cdots, M-1$

$$v_{k+1} = \alpha v_k - \epsilon \nabla f(x_k + \alpha v_k) \tag{26}$$
$$x_{k+1} = x_k + v_{k+1} \tag{27}$$

```
import numpy as np
def nesterov_method_2d(grad_func, x0,
    learning_rate=0.01, alpha=0.9, MaxIter=10):
    epsilon = learning_rate
    velocity = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*(x0 + alpha * velocity))
        velocity = alpha * velocity  - epsilon * grad
        x1 = x0 + velocity
        x0 = x1
    return x0
```

1. Start with initial x_0, v_0.

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

   - `x0` : initial guess for mimimizer
   - `epsilon` : learning rate

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

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

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

```
grad = grad_func(*(x0 + alpha * velocity))
```

   - `grad` : gradient of f at $x_k + \alpha v_k$

   (b) Update velocity, v_{k+1}

$$v_{k+1} = \alpha v_k - \epsilon \nabla f(x_k + \alpha v_k) \tag{28}$$

```
velocity = alpha * velocity  - epsilon * grad
```

   - `alpha` : constant for cumulative momentum storing ratio, α
   - `epsilon` : learning rate, ϵ

   (c) Update new position, x_{k+1}

$$x_{k+1} = x_k + v_{k+1} \tag{29}$$

```
x1 = x0 + velocity
x0 = x1
```

   - `velocity` : update size, v_{k+1}
   - `x1` : next position, x_{k+1}

Example 4.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{30}$$

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 $\nabla f = \begin{bmatrix} 6(x-2) \\ 2(y-2) \end{bmatrix}$

   ```
   grad_f = lambda x,y: np.array([6 * (x - 2), 2 * (y - 2)])
   ```

3. Tune parameters such as `x0, learning_rate, alpha, MaxIter`

4. Run Nesterov method!

```
def nesterov_method_2d(grad_func, w0, learning_rate=0.01, alpha=0.9, MaxIter=10):
    velocity = np.zeros_like(w0)
    for i in range(MaxIter):
        grad = grad_func(w0 + alpha * velocity)
        velocity = alpha * velocity  - learning_rate * grad
        w1 = w0 + velocity
        w0 = w1
    return w0

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

# Tune parameters(Use default values for MaxIter, learning_rate)
x0 = np.array([-2.0, -2.0])
xopt = nesterov_method_2d(f, grad_f, x0, learning_rate=0.2, MaxIter=75)

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

5 Adagrad in 2D

To solve the following minimization problem,

$$\min_x f(x) \tag{31}$$

in this section, we study Adagrad method(see [Duchi et al., 2011]) expressed as for given $x_0 \in \mathbf{R}^2$, do iteration for $k = 0, \cdots, M-1$

$$r_{k+1} = r_k + \nabla f(x_k) \odot \nabla f(x_k) \tag{32}$$

$$x_{k+1} = x_k - \frac{\epsilon}{\delta + \sqrt{r_{k+1}}} \odot \nabla f(x_k) \tag{33}$$

```python
import numpy as np
def adagrad_method_2d(grad_func, x0,
    learning_rate=0.01, delta=1E-7, MaxIter=10):
    epsilon = learning_rate
    r = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*x0)
        r = r  + grad * grad
        x1 = x0 - epsilon * grad / ( delta + np.sqrt(r) )
        x0 = x1
    return x0
```

1. Start with initial x_0, r_0.

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

    - `x0` : initial guess of minimizer
    - `epsilon` : learning rate, ϵ
    - `r` : variable for storing past gradient's information

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

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

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

    ```python
    grad = grad_func(*x0)
    ```

    - `grad` : gradient, $\nabla f(x_k)$, in ℓ^2-sense.

    (b) Store history of past gradients, $\nabla f(x_k)$,

    ```python
    r = r  + grad * grad
    ```

    - `r` : gradient, $\nabla f(x_k)$

    (c) Calculate search direction and update.

    ```python
    x1 = x0 - epsilon * grad / ( delta + np.sqrt(r) )
    ```

    - `delta` : constant for numerical stability
    - `grad / ( delta + np.sqrt(r) )` : adaptive weights for `Adagrad`

    (d) Update new position, x_{k+1}

    ```python
    x0 = x1
    ```

Example 5.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{34}$$

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 $\nabla f = \begin{bmatrix} 6(x-2) \\ 2(y-2) \end{bmatrix}$

```
grad_f = lambda x,y: np.array([6 * (x - 2), 2 * (y - 2)])
```

3. Tune parameters such as `x0, delta, learning_rate, MaxIter`

4. Run Adagrad method!

```
def adagrad_method_2d(grad_func, x0,
    learning_rate=0.01, delta=1E-7, MaxIter=10):
    epsilon = learning_rate
    r = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*x0)
        r = r  + grad * grad
        x1 = x0 - epsilon * grad / ( delta + np.sqrt(r) )
        x0 = x1
    return x0

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

# Tune parameters(Use default values for MaxIter, learning_rate)
x0 = np.array([-2.0, -2.0])
xopt = adagrad_method_2d(grad_f, x0, learning_rate=5, MaxIter=25)

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

6 RMSProp in 2D

To solve the following minimization problem,

$$\min_x f(x) \tag{35}$$

in this section, we study RMSProp(see [Hinton et al., 2012]) method expressed as for given $x_0 \in \mathbf{R}^2$, do iteration for $k = 0, \cdots, M-1$

$$r_{k+1} = \rho r_k + (1-\rho)\nabla f(x_k) \odot \nabla f(x_k) \tag{36}$$

$$x_{k+1} = x_k - \frac{\epsilon}{\sqrt{\delta + r_{k+1}}} \odot \nabla f(x_k) \tag{37}$$

```
import numpy as np
def rmsprop_method_2d(grad_func, x0,
    learning_rate=0.01, delta=1E-6, rho=0.9, MaxIter=10):
    epsilon = learning_rate
    r = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*x0)
        r = rho * r  + (1 - rho) * (grad * grad)
        x1 = x0 - epsilon  * grad / np.sqrt(delta + r)
        x0 = x1
    return x0
```

1. Start with initial x_0, r_0.

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

   - `x0` : initial guess of minimizer
   - `epsilon` : learning rate, ϵ
   - `r` : variable for storing past gradient's information

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

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

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

   ```
   grad = grad_func(*x0)
   ```

   - `grad` : gradient, $\nabla f(x_k)$, in ℓ^2-sense.

   (b) Store short(ρ) history of past gradients, $\nabla f(x_k)$,

   ```
   r = rho * r  + (1 - rho) * (grad * grad)
   ```

   - `rho` : memory coefficient who controls how long it remember history of gradients
   - `r` : gradient, $\nabla f(x_k)$

   (c) Calculate search direction and update.

   ```
   x1 = x0 - epsilon  * grad / np.sqrt(delta + r)
   ```

   - `delta` : constant for numerical division stability
   - `grad / np.sqrt(delta + r)` : adaptive weights for `Rmsprop`

   (d) Update new position, x_{k+1}

   ```
   x0 = x1
   ```

Example 6.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{38}$$

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 $\nabla f = \begin{bmatrix} 6(x-2) \\ 2(y-2) \end{bmatrix}$

```
grad_f = lambda x,y: np.array([6 * (x - 2), 2 * (y - 2)])
```

3. Tune parameters such as `x0, learning_rate, delta, rho, MaxIter`

4. Run RMSProp method!

```
def rmsprop_method_2d(grad_func, x0,
    learning_rate=0.01, delta=1E-6, rho=0.9, MaxIter=10):
    epsilon = learning_rate
    r = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*x0)
        r = rho * r  + (1 - rho) * (grad * grad)
        x1 = x0 - epsilon  * grad / np.sqrt(delta + r)
        x0 = x1
    return x0

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

# Tune parameters(Use default values for MaxIter, learning_rate)
x0 = np.array([-2.0, -2.0])
xopt = rmsprop_method_2d(grad_f, x0, learning_rate=0.5, MaxIter=25)

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

7 Adam in 2D

To solve the following minimization problem,

$$\min_x f(x) \tag{39}$$

in this section, we study Adam method(see [Kingma and Ba, 2014]) expressed as for given $x_0 \in \mathbf{R}^2$, do iteration for $k = 0, \cdots, M-1$

$$s_{k+1} = \rho_1 s_k + (1-\rho_1)\nabla f(x_k) \tag{40}$$
$$r_{k+1} = \rho_2 r_k + (1-\rho_2)\nabla f(x_k) \odot \nabla f(x_k) \tag{41}$$
$$\hat{s}_{k+1} = \frac{s_{k+1}}{1-\rho_1^t} \tag{42}$$
$$\hat{r}_{k+1} = \frac{r_{k+1}}{1-\rho_2^t} \tag{43}$$
$$x_{k+1} = x_k - \frac{\epsilon}{\sqrt{\hat{r}_{k+1}} + \delta}\hat{s}_{k+1} \tag{44}$$

```
import numpy as np
def adam_method_2d(grad_func, x0,
    learning_rate=0.001, delta=1E-8, rho1=0.9, rho2=0.999, MaxIter=10):
    epsilon = learning_rate
    s = np.zeros_like(x0)
    r = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*x0)
        s = rho1 * s  + (1 - rho1) * grad
        r = rho2 * r  + (1 - rho2) * (grad * grad)
        shat = s / (1. - rho1 ** (i+1))
        rhat = r / (1. - rho2 ** (i+1))
        x1 = x0 - epsilon  * shat / (delta + np.sqrt(rhat))
        x0 = x1
    return x0
```

1. Start with initial x_0, r_0, and s_0.

   ```
   import numpy as np
   x0 = np.array([-2.0, -2.0])
   epsilon = learning_rate
   s = np.zeros_like(x0)
   r = np.zeros_like(x0)
   ```

   - `x0` : initial guess of minimizer
   - `epsilon` : learning rate, ϵ
   - `s` : variable for storing past the first moment's information
   - `r` : variable for storing past the second moment's information

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

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

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

   ```
   grad = grad_func(*x0)
   ```

   (b) Compute biased first and second moment, s_{k+1}, r_{k+1}

   ```
   s = rho1 * s  + (1 - rho1) * grad
   r = rho2 * r  + (1 - rho2) * (grad * grad)
   ```

   - `rho1` : variable for storing past gradient's information
   - `rho2` : variable for storing past gradient's information

   (c) Correct bias in first and second moment

   ```
   shat = s / (1. - rho1 ** (i+1))
   rhat = r / (1. - rho2 ** (i+1))
   ```

- `shat` : corrected bias in the first moment
- `rhat` : corrected bias in the second moment

(d) Compute search direction, $\frac{\hat{s}}{\delta+\sqrt{\hat{r}_{k+1}}}$

```
x1 = x0 - epsilon  * shat / (delta + np.sqrt(rhat))
```

- `shat / (delta + np.sqrt(rhat)` : variable for storing past gradient's information

(e) Udate new position, x_{k+1}

```
x0 = x1
```

Example 7.

$$\min_{x,y} \left[3(x-2)^2 + (y-2)^2\right] \tag{45}$$

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 $\nabla f = \begin{bmatrix} 6(x-2) \\ 2(y-2) \end{bmatrix}$

```
grad_f = lambda x,y: np.array([6 * (x - 2), 2 * (y - 2)])
```

3. Tune parameters such as `x0, learning_rate, delta, rho1, rho2, MaxIter`

4. Run Adam method!

```
def adam_method_2d(grad_func, x0,
    learning_rate=0.001, delta=1E-8, rho1=0.9, rho2=0.999, MaxIter=10):
    epsilon = learning_rate
    s = np.zeros_like(x0)
    r = np.zeros_like(x0)
    for i in range(MaxIter):
        grad = grad_func(*x0)
        s = rho1 * s  + (1 - rho1) * grad
        r = rho2 * r  + (1 - rho2) * (grad * grad)
        shat = s / (1. - rho1 ** (i+1))
        rhat = r / (1. - rho2 ** (i+1))
        x1 = x0 - epsilon  * shat / (delta + np.sqrt(rhat))
        x0 = x1
    return x0

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

# Tune parameters(Use default values for MaxIter, learning_rate)
x0 = np.array([-2.0, -2.0])
xopt = adam_method_2d(grad_f, x0, learning_rate=2, MaxIter=400)

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

Comment

If you have further interests in optimization for deep learning, [Goodfellow et al., 2016] is a good book explaining deep learning in view of mathematics.

References

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Hinton et al., 2012] Hinton, G., Srivastava, N., and Swersky, K. (2012). Rmsprop: Divide the gradient by a running average of its recent magnitude. *Neural networks for machine learning, Coursera lecture 6e*.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Nesterov, 1983] Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376.