Algoritmo del gradiente conjugado

Introducción

El **algoritmo del gradiente conjugado** es parte de una clase de métodos llamados **métodos de dirección conjugada**. Estos métodos suelen tener un mejor rendimiento que el método del descenso del gradiente, pero no tan bueno como el método de Newton, por eso se pueden considerar de un nivel intermedio entre ambos métodos. La ventaja que tiene el algoritmo del gradiente conjugado respecto al método de Newton es que no se requiere evaluar la matriz ***Hessiana*** de la función.

Definición:

Dada una matriz Q con entradas en los reales simétrica de $n \times n$. Las direcciones $\mathbf{d}_0, \mathbf{d}_1, \dots, \mathbf{d}_k \in \mathbb{R}^n$ se dicen **Q-conjugadas** si para cualesquiera $i \neq j$ se cumple que $\mathbf{d}_i^T Q \mathbf{d}_j = 0$.

Lema

Sea Q una matriz simétrica definida positiva de tamaño $n \times n$. Si las direcciones $\mathbf{d}_0, \mathbf{d}_1, \dots, \mathbf{d}_k \in \mathbb{R}^n$, con $k \leq n-1$, son no nulas y **Q-conjugadas**, entonces son linealmente independientes.

Demostración.
Sean $\alpha_0, \dots, \alpha_k$ escalares tales que:

$$\alpha_0 \mathbf{d}_0 + \alpha_1 \mathbf{d}_1 + \cdots + \alpha_k \mathbf{d}_k = \mathbf{0} \in \mathbb{R}^n.$$

Notemos que para $j = 0, 1, \dots, k$:

$$0 = \mathbf{d}_j^T Q \mathbf{0} = \mathbf{d}_j^T Q(\alpha_0 \mathbf{d}_0 + \alpha_1 \mathbf{d}_1 + \cdots + \alpha_k \mathbf{d}_k) = \alpha_j \mathbf{d}^{(j)\top} Q \mathbf{d}^{(j)}$$

porque todos los demás términos $\mathbf{d}_j^T Q \mathbf{d}_i = 0$ para $i \neq j$, por la Q-conjugación.
Pero $Q = Q^T > 0$ y $\mathbf{d}_j \neq \mathbf{0}$; por lo tanto, $\alpha_j = 0$, para $j = 0, 1, \ldots, k$.

Por lo tanto $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_k$, con $k \leq n-1$, son linealmente independientes.

El algoritmo de la dirección conjugada

Primero nos enfocaremos en minimizar funciones $f : \mathbb{R}^n \to \mathbb{R}$ de la forma

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q \mathbf{x} - \mathbf{x}^T \mathbf{b}$$

con $Q = Q^T > 0$.

Lema

Si $g : \mathbb{R}^n \to \mathbb{R}$ es tal que $g(\mathbf{x}) = \mathbf{x}^T Q \mathbf{x}$ con Q una matriz de $n \times n$ con coeficientes reales, entonces $Dg(\mathbf{x}) = (Q + Q^T)\mathbf{x}$.

Demostración.

Notemos que

$$\mathbf{x}^T Q \mathbf{x} = \sum_{j=1}^{n} x_j \sum_{i=1}^{n} x_i Q_{ji}$$

La derivada con respecto a la variable k-ésima es (por la regla del producto):

$$\frac{d\left(\mathbf{x}^T Q \mathbf{x}\right)}{dx_k} = \sum_{j=1}^{n} \frac{dx_j}{dx_k} \sum_{i=1}^{n} x_i Q_{ji} + \sum_{j=1}^{n} x_j \sum_{i=1}^{n} \frac{dx_i}{dx_k} Q_{ji}$$

$$= \sum_{i=1}^{n} x_i Q_{ki} + \sum_{j=1}^{n} x_j Q_{jk}$$

Si acomodamos estas derivadas en un vector columna, obtenemos:

$$\begin{bmatrix} \sum_{i=1}^{n} x_i Q_{1i} + \sum_{j=1}^{n} x_j Q_{j1} \\ \sum_{i=1}^{n} x_i Q_{2i} + \sum_{j=1}^{n} x_j Q_{j2} \\ \vdots \\ \sum_{i=1}^{n} x_i Q_{ni} + \sum_{j=1}^{n} x_j Q_{jn} \end{bmatrix} = Q\mathbf{x} + \left(\mathbf{x}^T Q\right)^T = (Q + Q^T)\mathbf{x}$$

Lema

Si $f : \mathbb{R}^n \to \mathbb{R}$ es de la forma $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b}$ con $Q = Q^T > 0$, entonces $Df(\mathbf{x}) = Q\mathbf{x} - \mathbf{b}$.

Demostración.

$$Df(\mathbf{x}) = D(\frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b}) = \frac{1}{2}D(\mathbf{x}^T Q\mathbf{x}) - D(\mathbf{x}^T\mathbf{b})$$
$$= \frac{1}{2}(Q + Q^T)\mathbf{x} - \mathbf{b} = \frac{1}{2}(Q + Q)\mathbf{x} - \mathbf{b} = Q\mathbf{x} - \mathbf{b}$$

Lema

Si $f : \mathbb{R}^n \to \mathbb{R}$ es de la forma $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b}$ con $Q = Q^T > 0$, entonces $D^{(2)}f(\mathbf{x}) = Q$.

Demostración.

$$D^{(2)}f(\mathbf{x}) = D(Q\mathbf{x} - \mathbf{b}) = D(Q\mathbf{x}) - D(\mathbf{b}) = Q$$

porque $T : \mathbb{R}^n \to \mathbb{R}$ dada por $T(\mathbf{x}) = Q\mathbf{x}$ es lineal.

Teorema

Si $f : \mathbb{R}^n \to \mathbb{R}$ es de la forma $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b}$ con $Q = Q^T > 0$, entonces el problema de $\min f(\mathbf{x})$ se reduce a resolver el sistema $Q\mathbf{x} = b$.

Demostración.

Supongamos que $\mathbf{x}^* \in \mathbb{R}^n$ cumple que $Q\mathbf{x}^* = \mathbf{b}$, entonces $Df(\mathbf{x}^*) = Q\mathbf{x}^* - \mathbf{b} = \mathbf{b} - \mathbf{b} = \mathbf{0}$

Además, por el lema anterior $\text{Hess}f(\mathbf{x}^*) = Q$ es definida positiva, por lo tanto se cumplen las condiciones suficientes de segundo orden para que $\mathbf{x}^*$ sea un mínimo local.

Algoritmo Básico de Direcciones Conjugadas.
Dado un punto inicial $\mathbf{x}^{(0)}$ y direcciones **Q-conjugadas** $\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{n-1}$; para $k \geq 0$ construimos la sucesión de puntos:

$$\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)}) = Q\mathbf{x}^{(k)} - \mathbf{b},$$

$$\alpha_k = -\frac{\mathbf{g}^{(k)^T}\mathbf{d}_k}{\mathbf{d}_k^T Q \mathbf{d}_k},$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}_k.$$

El algoritmo del gradiente conjugado

El algoritmo de direcciones conjugadas es muy eficaz. Sin embargo tiene la desventaja de que para utilizar el algoritmo, necesitamos especificar las direcciones **Q-conjugadas**. El **algoritmo del gradiente conjugado** no utiliza direcciones conjugadas preespecificadas, sino que calcula las direcciones a medida que avanza el algoritmo. En cada etapa del algoritmo, la dirección se calcula como una combinación lineal de la dirección anterior y el gradiente actual, de tal manera que todas las direcciones son mutuamente **Q-conjugadas**.

El algoritmo del gradiente conjugado es el siguiente:

1. Sea $k := 0$; se selecciona el punto inicial $\mathbf{x}^{(0)}$.
2. Sea $\mathbf{g}^{(0)} = \nabla f(\mathbf{x}^{(0)})$.
   Si $\mathbf{g}^{(0)} = 0$, detenerse; de lo contrario, sea $\mathbf{d}_0 = -\mathbf{g}^{(0)}$.
3. $$\alpha_k = -\frac{\mathbf{g}^{(k)\top}\mathbf{d}_k}{\mathbf{d}_k^T Q \mathbf{d}_k}.$$
4. $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}_k.$$
5. $$\mathbf{g}^{(k+1)} = \nabla f(\mathbf{x}^{(k+1)}).$$

   Si $\mathbf{g}^{(k+1)} = 0$, detenerse.
6. $$\beta_k = \frac{\mathbf{g}^{(k+1)\top} Q \mathbf{d}_k}{\mathbf{d}_k^T Q \mathbf{d}_k}.$$
7. $$\mathbf{d}_{k+1} = -\mathbf{g}^{(k+1)} + \beta_k \mathbf{d}_k.$$
8. Sea $k := k + 1$; ir al paso 3.

> **Advertencia**
>
> Aunque en teoría se puede resolver una ecuación con el método del gradiente conjugado, esto no es muy eficiente, porque requiere más operaciones que otras formas como una descomposición de Cholesky o LDL. Además, la convergencia solo se cumple cuando todos los cálculos se hacen de forma exacta, sin errores de redondeo que se pueden presentar en programas de computadora al usar aritmética de punto flotante.

Implementación del algoritmo en Python:

La siguiente función de Python minimiza una función de la forma $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b}$ con $Q = Q^T > 0$, para eso pide como valores de entrada a la matriz $Q = Q^T > 0$ y el vector $\mathbf{b}$.

```
def minimize_quadratic_form(Q, b):
  grad = lambda x: Q @ x - b
  n = Q.shape[0] # Aprovechamos que la matriz es cuadrada
  x_k = np.random.random((n,1))
  ret_list = [x_k]
  g_k = grad(x_k)
  d_k = - g_k
  for _ in range(n+2):
    alpha_k = - (g_k.T @ d_k) / (d_k.T @ (Q @ d_k))
    x_k = x_k + alpha_k*d_k
    ret_list.append(x_k)
    g_k = grad(x_k)
    beta_k = (g_k.T @ d_k) / (d_k.T @ (Q @ d_k))
    d_k = - g_k + beta_k*d_k

  return ret_list
```

Ejemplo 1

Consideremos los siguientes valores de entrada:

```
Q1 = np.array([[3, 0, 1],
               [0, 4, 2],
               [1, 2, 3]])
b1 = np.array([[3],[0],[1]])
```

Tenemos el siguiente valor óptimo, mientras que valor exacto es $(1, 0, 0)^T$

```
array([[ 0.98016253],
       [-0.02963407],
       [ 0.02856911]])
```

> **Precaución**
>
> El valor obtenido puede variar debido a que se toma el punto inicial al azar.

Ejemplo 2

Consideremos ahora los siguientes valores de entrada:

```
Q2 = np.array([[4, 2],
               [2, 2]])

b2 = np.array([[-1],[1]])

iterations = minimize_quadratic_form(Q2, b2)
```

Nos dio un valor de $(-0.97350862, 1.4902551)^T$ mientras que el valor exacto del mínimo es $(-1, 1.5)^T$

El algoritmo del gradiente conjugado para funciones no cuadráticas

Notemos que la versión del algoritmo del gradiente conjugado que tenemos hasta ahora es un método de direcciones conjugadas, y por lo tanto, minimiza una función cuadrática definida positiva de n variables en n pasos. Afortunadamente el método puede extenderse a funciones no lineales generales tomando $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} - \mathbf{x}^T\mathbf{b}$ como una aproximación por el **polinomio de Taylor multivariable** de segundo orden de la función objetivo. Cerca de la solución, tales funciones se comportan aproximadamente como cuadráticas.

Para una función cuadrática, la matriz Q, es la matriz **Hessiana** de la cuadrática, es decir que es constante. Sin embargo, para una función no lineal general, la matriz Hessiana debe reevaluarse en cada iteración del algoritmo, lo que es muy costoso computacionalmente hablando. Por lo tanto, implementaremos una versión eficiente del algoritmo del gradiente conjugado que elimine la evaluación de la matriz Hessiana en cada paso.

Observemos que Q aparece únicamente en el cálculo de los escalares α_k y β_k. Dado que α_k es el minimizador de $\phi(\alpha) = f(\mathbf{x}^{(k)} + \alpha\mathbf{d}_k)$, basta con encontrar un α_k que nos produzca un valor menor respecto a la iteración antetior y eso lo podemos hacer evaluando directamente la función f, sin necesidad de evaluar la Hessiana.

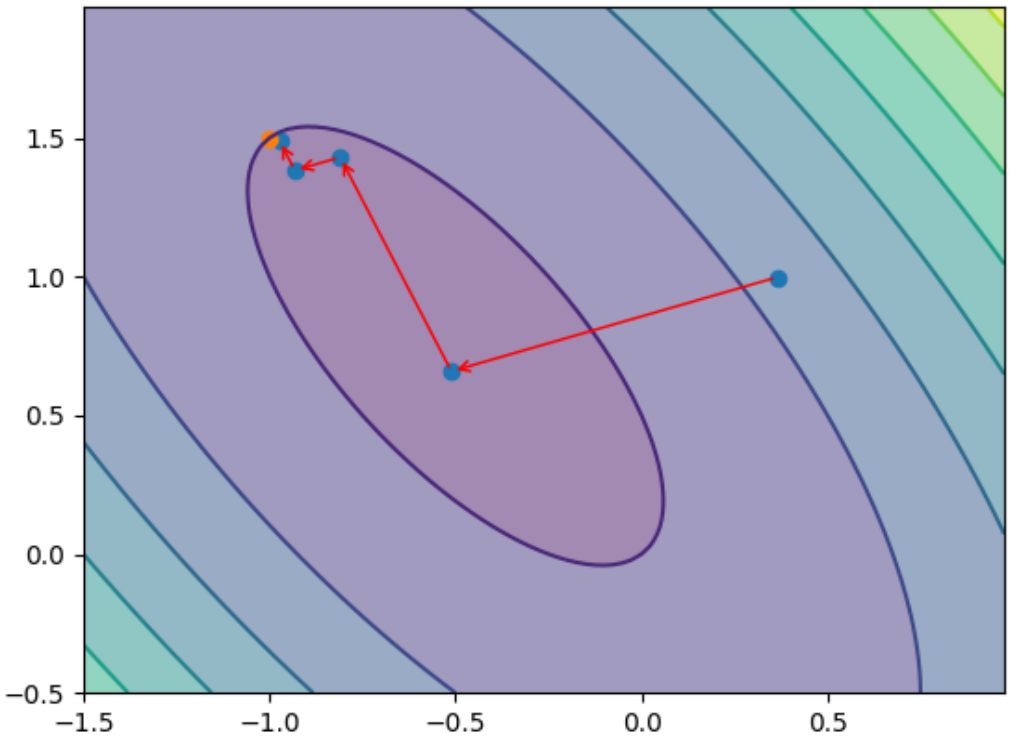

Figura 1: gráfica

Para β_k podemos usar la siguiente expresión:

$$\beta_k = \max\left\{0,\ \frac{\mathbf{g}^{(k+1)\top}\left[\mathbf{g}^{(k+1)} - \mathbf{g}^{(k)}\right]}{\mathbf{g}^{(k)\top}\mathbf{g}^{(k)}}\right\}.$$

Implementación del algoritmo en Python:

La siguiente función de Python minimiza una función de la forma $f : \mathbb{R}^n \to \mathbb{R}$, para eso pide como valores de entrada a la función f, su gradiente ∇f, el número de iteraciones a realizar y el punto inicial.

```
def minimize_nonquadratic_form(f, grad, num_iters, x_0):
  x_k = x_0
  ret_list = [x_k]
  g_k_1 = grad(x_k)
  d_k = - g_k_1
  for _ in range(num_iters):
    alpha_k = 1
    # Buscamos un alpha adecuado que nos produzca una mejor iteración
    while f(x_k + alpha_k*d_k) > f(x_k):
       alpha_k = alpha_k/2
    x_k = x_k + alpha_k*d_k
    ret_list.append(x_k)
    g_k = grad(x_k)
    beta_k = np.max([(g_k.T @ (g_k - g_k_1)) / (g_k_1.T @ g_k_1), 0])
    d_k = - g_k + beta_k*d_k
    g_k_1 = g_k

  return ret_list
```

Ejemplo 3

Consideramos la función $f(x, y) = \exp(x^2 + y^2)$ que tiene un mínimo en $(0, 0)^T$.

```
def f1(x):
  return np.exp(x[0]**2 +  x[1]**2)

def grad1(x):
  return np.array([2*x[0]*f1(x), 2*x[1]*f1(x)])

iterations = minimize_nonquadratic_form(f1, grad1, 10, np.array([1, 1.5]))
```

Con el código anterior se obtuvo un valor `array([-0.00877747, -0.01316621])`.

Referencias

1. **AN INTRODUCTION TO OPTIMIZATION**. Edwin K. P. Chong y Stanislaw H. Zak. Cuarta edición. Publicado por *John Wiley & Sons, Inc.* Páginas 175-188.
2. **Continuous Optimisation**. Markus Grasmair. Notas disponibles en internet. Consultadas el 22/06/2025. Páginas 35-39.