403 MÉTODO DE NEWTON

October 17, 2025

1 Método de Newton

1.1 Aproximación mediante la serie de Taylor

La derivación formal del método de Newton se sustenta en la aproximación de una función mendiante su **serie de Taylor**. Este principio permite transformar un problema no lineal complejo, como es la búsqueda de raíces, en una secuencia de problemas lineales que son computacionalmente más tratables.

La **serie de Taylor** permite representar una función $f(x)$ suficientemente diferenciable en el entorno de un punto $x = x_n$ como una suma infinita de términos basados en sus derivadas en dicho punto:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2!}(x - x_n)^2 + \frac{f'''(x_n)}{3!}(x - x_n)^3 + \cdots$$

Para los propositos del método de Newton, se trunca esta serie después del término de primer orden, lo que resulta en la **aproximación lineal** de la función $f(x)$ en la vecindad de x_n:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n)$$

Esta aproximación asume que, en un intervalo suficientemente pequeño alrededor de x_n, el comportamiento de la función puede ser modelado con precisión por la línea tangente a la curva en ese punto.

Para la búqueda de raíces, el objetivo es encontrar un valor de x para la cual $f(x) = 0$. Utilizando la aproximación lineal, el problema se simplifica a resolver la siguiente ecuación:

$$0 = f(x_n) + f'(x_n)(x - x_n)$$

Al resolver esta ecuación para x, obtenemos una nueva y mejorada aproximación de la raíz, que denotamos como x_{n+1}. Partiendo de la ecuación anterior, donde x se convierte en x_{n+1}:

$$0 = f(x_n) + f'(x_n)(x_{n+1} - x_n)$$

Se despeja x_{n+1} para obtener la **formula iterativa del método de Newton**:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Esta expresión define una secuencia de aproximaciones $x_0, x_1, x_2, ...$ que, bajo condiciones favorables, converge a la raíz buscada de la función.

1.2 Adaptación para problemas de optimización

El método de Newton, tradicionalmente usado para encontrar raíces, puede adaptarse de manera eficaz para resolver problemas de optimización, es decir, para encontrar el punto que minimiza o maximiza una función $f(x)$.

El proceso para esta adaptación se resume en los siguientes pasos:

- **Condición de optimalidad:** El primer paso es reconocer que un punto óptimo (mínimo o máximo) de una función $f(x)$ ocurre donde su primera derivada es cero. Matemáticamente, si x^* es un punto óptimo, entonces:

$$f'(x^*) = 0$$

- **Transformación a un problema de raíces:** Esta condición permite transformar el problema de optimización en un problema de búsqueda de raíces. En lugar de buscar el óptimo de $f(x)$, ahora buscamos la raíz de su función derivada, $f'(x) = 0$.
- **Aplicación del método de Newton:** Para encontrar esta raíz, definimos una nueva función $g(x) = f'(x)$. Ahora podemos aplicar la fórmula iterativa estándar del método de Newton a $g(x)$:

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)}$$

- **Fórmula de Newton para optimización:** Para obtener la fórmula final, simplemente sustituimos $g(x)$ por $f'(x)$ y, por lo tanto, $g'(x)$ por la segunda derivada, $f''(x)$. Esto nos da la **fórmula iterativa de Newton para optimización**:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Para generalizar el método a casos multivariados, es fundamental conocer sus componentes principales:

- **Función objetivo:** La función $f(\mathbf{x})$ que se quiere optimizar. El objetivo es encontrar el valor de $\mathbf{x}$ que la hace mínima o máxima.
- **Primera derivada (vector gradiente):** Representado como $\nabla f(\mathbf{x})$, es un vector que indica la dirección de máximo crecimiento de la función en un punto $\mathbf{x}$. Apunta "cuesta arriba" y su magnitud representa la pendiente en esa dirección. En la optimización, lo usamos para saber hacia dónde movernos.

- **Segunda derivada (matriz Hessiana):** Simbolizada como $\mathbf{H}_f(\mathbf{x})$, esta matriz contiene las segundas derivadas parciales de la función. Nos informa sobre la **curvatura** de la función (si es cóncava o convexa) en un punto, lo cual es crucial para determinar si estamos cerca de un mínimo o un máximo.
- **Proceso iterativo:** El algoritmo, comienza con una suposición inicial y la refina sucesivamente a través de una fórmula de actualización, acercándose cada vez más al punto óptimo.

El método para minimizar una función $f(\mathbf{x})$ se puede describir en los siguientes tres pasos:

1. **Inicialización:** Se elige un punto de partida inicial $\mathbf{x}_0$. Una buena elección, cercana a la solución real, puede acelerar significativamente el proceso.
2. **Iteración y actualización:** Para cada paso $k = 0, 1, 2, ...$, se realiza el siguiente cálculo para obtener una mejor aproximación $\mathbf{x}_{k+1}$: se calcula el gradiente $\nabla f(\mathbf{x}_k)$, se calcula la matriz Hessiana $\mathbf{H}_f(\mathbf{x}_k)$ y su inversa, y se actualiza la posición usando la fórmula de Newton:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}_f^{-1}(\mathbf{x}_k)\nabla f(\mathbf{x}_k)$$

3. **Criterio de convergencia:** Se finalizan las iteraciones si se cumple algún criterio de convergencia. Si es así, se retorna $\mathbf{x}_{k+1}$ como solución aproximada.

El método de Newton converge rápidamente hacia el mínimo local cuando la función es convexa y su Hessiana es positiva definida. Sin embargo, puede no funcionar bien si la Hessiana no es definida positiva o si el punto inicial está lejos de la solución óptima.

1.3 Ejemplos

[1]:
```
# -------------------------
# Configuraciones iniciales
# -------------------------
import numpy as np
import matplotlib.pyplot as plt
resolucion = 600
```

[2]:
```
# ------------------------------
# Algoritmo de Newton univariado
# ------------------------------
def newton_univariado(f, grad_f, hess_f, x_actual, max_iter = 100, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - grad_f(x_actual) / hess_f(x_actual)
        x_historial = np.append(x_historial, x_nuevo)
        criterio_1 = np.linalg.norm(grad_f(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if criterio_1 < tol or criterio_2 < tol:
            break
        x_actual = x_nuevo
    f_optimo = f(x_nuevo)
    iteraciones = len(x_historial)
```

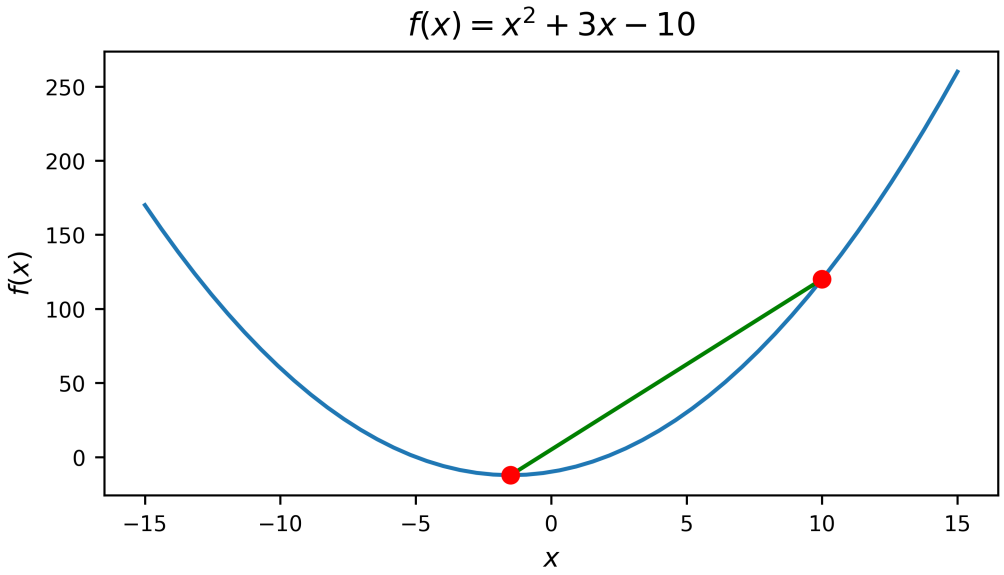
```
    return x_nuevo, f_optimo, iteraciones, x_historial
```

[3]:
```
# ---------------------------------
# Encontrar el mínimo de la función:
# f(x) = x ** 2 + 3 * x - 10
# ---------------------------------

# Definición de la función objetivo, gradiente y Hessiana
def polinomio(x):
    return x ** 2 + 3 * x - 10

def grad_polinomio(x):
    return 2 * x + 3

def hess_polinomio(x):
    return 2

# Definición del punto inicial
x_actual = 10

# Ejecución del algoritmo
r_polinomio = newton_univariado(polinomio, grad_polinomio, hess_polinomio,␣
 ↪x_actual)

# Resultados
print("x =", r_polinomio[0])
print("f(x) =", r_polinomio[1])
print("Número de iteraciones:", r_polinomio[2])
```

```
x = -1.5
f(x) = -12.25
Número de iteraciones: 2
```

[4]:
```
# Graficar la función
x = np.linspace(-15, 15, 50)
y = polinomio(x)

plt.figure(figsize=(6, 3), dpi = resolucion)
plt.plot(x, y)
plt.title("$f(x) = x^2 + 3x - 10$")
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.tick_params(labelsize = 8)

# Puntos de trayectoria
plt.plot(r_polinomio[3], polinomio(r_polinomio[3]), "g-")
plt.plot(r_polinomio[3], polinomio(r_polinomio[3]), "ro")
```

```
plt.show()
```



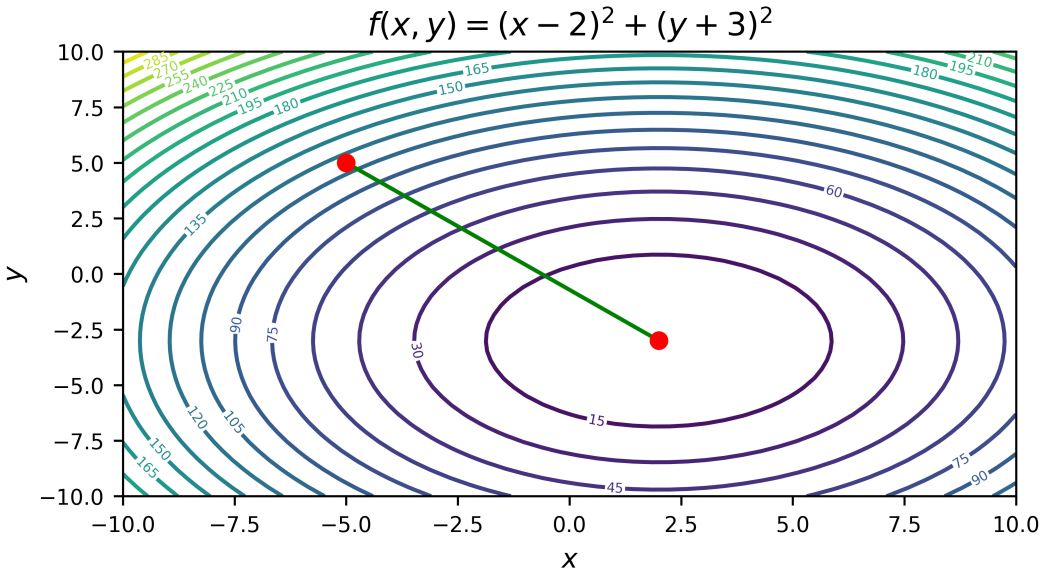
[5]:
```
# --------------------------------
# Algoritmo de Newton multivariado
# --------------------------------
def newton_multivariado(f, grad_f, hess_f, x_actual, max_iter = 100, tol =␣
 ↪1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - np.linalg.inv(hess_f(x_actual)) @ grad_f(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_f(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if criterio_1 < tol or criterio_2 < tol:
            break
        x_actual = x_nuevo
    f_optimo = f(x_nuevo)
    iteraciones = len(x_historial)
    return x_nuevo, f_optimo, iteraciones, x_historial
```

[6]:
```
# -----------------------------------
# Encontrar el mínimo de la función:
# f(x, y) = (x - 2) ** 2 + (y + 3) ** 2
# -----------------------------------

# Definición de la función objetivo, gradiente y Hessiana
def paraboloide(x):
```

```
    return (x[0] - 2) ** 2 + (x[1] + 3) ** 2

def grad_paraboloide(x):
    dx = 2 * (x[0] - 2)
    dy = 2 * (x[1] + 3)
    return np.array([dx, dy])

def hess_paraboloide(x):
    return np.array([[2, 0], [0, 2]])

# Definición del punto inicial
x_actual = np.array([-5, 5])

# Ejecución del algoritmo
r_paraboloide = newton_multivariado(paraboloide, grad_paraboloide,␣
 ↪hess_paraboloide, x_actual)

# Resultados
print("x =", r_paraboloide[0])
print("f(x) =", r_paraboloide[1])
print("Número de iteraciones:", r_paraboloide[2])
```

```
x = [ 2. -3.]
f(x) = 0.0
Número de iteraciones: 2
```

[7]:
```
# Graficar la función
x = np.linspace(-10, 10, 50)
y = np.linspace(-10, 10, 50)
X, Y = np.meshgrid(x, y)
Z = paraboloide([X, Y])

plt.figure(figsize = (6, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")
plt.clabel(contour, inline = True, fontsize = 5)
plt.title("$f(x, y) = (x - 2)^2 + (y + 3)^2$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.tick_params(labelsize = 8)

# Puntos de trayectoria
plt.plot(r_paraboloide[3][:, 0], r_paraboloide[3][:, 1], "g-")
plt.plot(r_paraboloide[3][:, 0], r_paraboloide[3][:, 1], "ro")
plt.show()
```

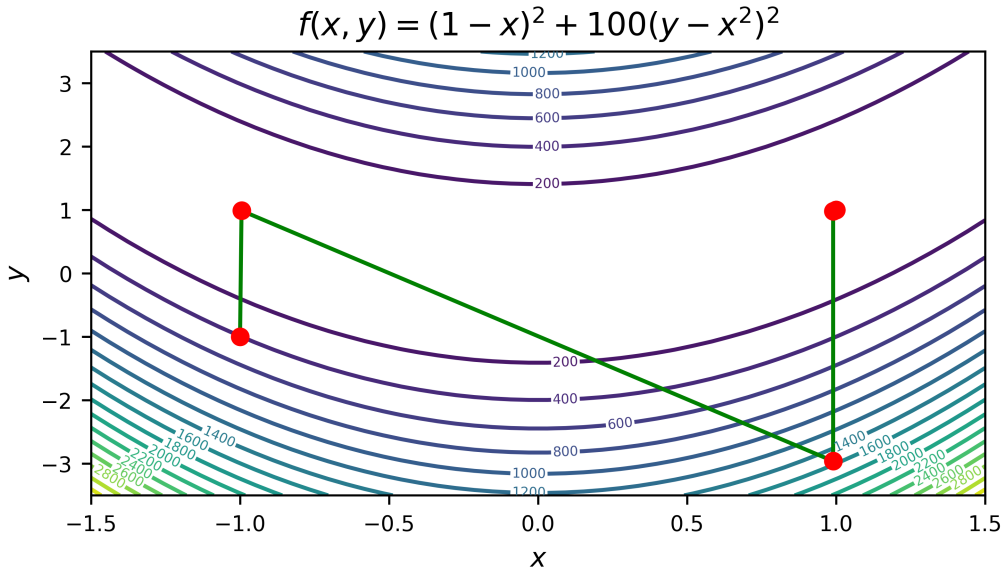


[8]:
```python
# ------------------------------------------------
# Encontrar el mínimo de la función de Rosenbrock:
# f(x, y) = (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
# ------------------------------------------------

# Definimos la función objetivo, gradiente y Hessiana
def rosenbrock(x):
    return (1 - x[0]) ** 2 + 100 * (x[1] - x[0] ** 2) ** 2

def grad_rosenbrock(x):
    dx = -2 * (1 - x[0]) - 400 * x[0] * (x[1] - x[0] ** 2)
    dy = 200 * (x[1] - x[0] ** 2)
    return np.array([dx, dy])

def hess_rosenbrock(x):
    dxx = 2 - 400 * (x[1] - 3 * x[0] ** 2)
    dyy = 200
    dyx = -400 * x[0]
    return np.array([[dxx, dyx], [dyx, dyy]])

# Definición del punto inicial
x_actual = np.array([-1, -1])

# Ejecución del algoritmo
r_rosenbrock = newton_multivariado(rosenbrock, grad_rosenbrock,␣
 ↪hess_rosenbrock, x_actual)
```

```
# Resultados
print("x =", r_rosenbrock[0])
print("f(x) =", r_rosenbrock[1])
print("Número de iteraciones =", r_rosenbrock[2])
```

```
x = [1. 1.]
f(x) = 3.4781872520856105e-23
Número de iteraciones = 6
```

[9]:
```
# Graficar la función
x = np.linspace(-1.5, 1.5, 50)
y = np.linspace(-3.5, 3.5, 50)
X, Y = np.meshgrid(x, y)
Z = rosenbrock([X, Y])

plt.figure(figsize = (6, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")
plt.clabel(contour, inline = True, fontsize = 5)
plt.title("$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.tick_params(labelsize = 8)

# Puntos de trayectoria
plt.plot(r_rosenbrock[3][:, 0], r_rosenbrock[3][:, 1], "g-")
plt.plot(r_rosenbrock[3][:, 0], r_rosenbrock[3][:, 1], "ro")
plt.show()
```



1.4 Consideraciones prácticas

Para implementar el método de Newton de manera efectiva y robusta, es fundamental tener en cuenta la siguiente consideración práctica. Además de establecer criterios de convergencia y la elección del punto inicial, el método de Newton requiere que la matriz Hessiana sea **positiva definida** para aseguerar que el paso de actualización en la dirección correcta (es decir, hacia un mínimo).

Si el Hessiano no es positivo definido, puede ser necesario modificarlo. Una técnica común es la **regularización**, donde se agrega una pequeña constante diagonal a la matriz Hessiana. Esto se logra agregando una matriz diagonal positiva, como $\mathbf{H}_f(\mathbf{x}_k) + c\mathbf{I}$, donde c es una constante positiva suficientemente grande.

[10]:
```python
def newton(f, grad_f, hess_f, x_actual, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        hessian = hess_f(x_actual)
        eigenvalores = np.linalg.eigvals(hessian)
        if np.any(eigenvalores <= 0):
            hessian = hessian + (np.abs(np.min(eigenvalores)) + 0.1) + np.
 ↪identity(2)
        x_nuevo = x_actual - np.linalg.inv(hessian) @ grad_f(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_f(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if criterio_1 < tol or criterio_2 < tol:
            break
        x_actual = x_nuevo
    f_optimo = f(x_nuevo)
    iteraciones = len(x_historial)
    return x_nuevo, f_optimo, iteraciones, x_historial
```

1.5 Aplicaciones: regresión polinomial

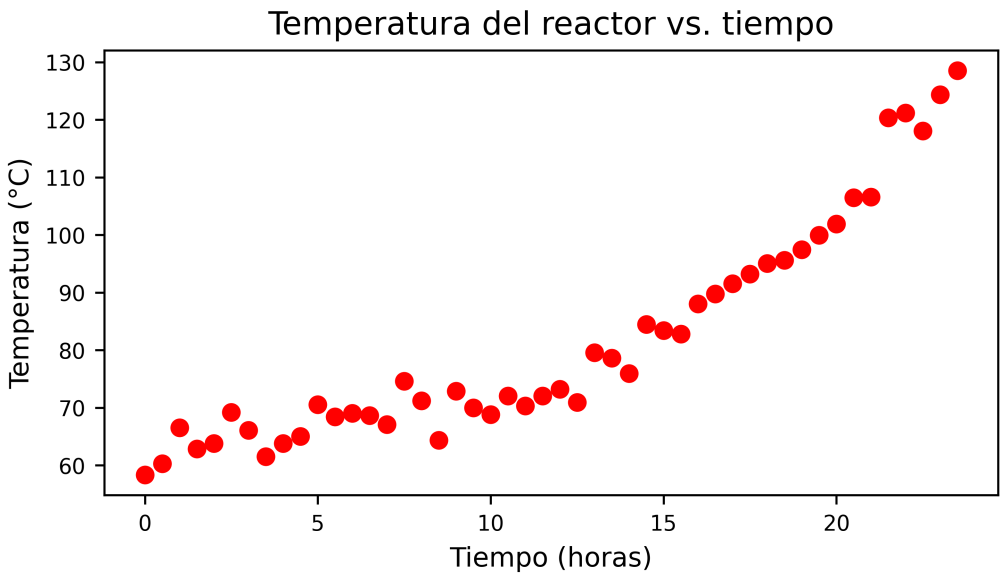
Consideremos un escenario donde se monitorea la temperatura de un reactor químico durante las primeras 24 horas de un proceso de síntesis. Los sensores registran mediciones cada media hora, pero debido a fluctuaciones del proceso y ruido instrumental, los datos presentan variabilidad. Se requiere modelar la tendencia temporal de la temperatura para:

- Interpolar valores en momentos no medidos
- Predecir el comportamiento futuro a corto plazo
- Identificar patrones anómalos en el proceso

Buscamos ajustar un polinomio de grado 3:

$$p(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

[11]:
```
# Datos
x = np.array([0.0, 0.5, 1.0, 1.5, 2.0,
              2.5, 3.0, 3.5, 4.0, 4.5,
              5.0, 5.5, 6.0, 6.5, 7.0,
              7.5, 8.0, 8.5, 9.0, 9.5,
              10.0, 10.5, 11.0, 11.5, 12.0,
              12.5, 13.0, 13.5, 14.0, 14.5,
              15.0, 15.5, 16.0, 16.5, 17.0,
              17.5, 18.0, 18.5, 19.0, 19.5,
              20.0, 20.5, 21.0, 21.5, 22.0,
              22.5, 23.0, 23.5])

y = np.array([58.319, 60.268, 66.515, 62.859, 63.780,
              69.223, 66.096, 61.508, 63.795, 65.041,
              70.547, 68.434, 69.026, 68.621, 67.089,
              74.595, 71.222, 64.345, 72.895, 69.956,
              68.797, 72.022, 70.331, 72.019, 73.197,
              70.956, 79.556, 78.621, 75.962, 84.457,
              83.404, 82.787, 88.029, 89.781, 91.552,
              93.238, 95.070, 95.616, 97.443, 99.950,
              101.916, 106.470, 106.583, 120.370,
              121.176, 118.084, 124.364, 128.518])

# Diagrama de dispersión
plt.figure(figsize = (6, 3), dpi = resolucion)
plt.plot(x, y, "ro")
plt.title("Temperatura del reactor vs. tiempo")
plt.xlabel("Tiempo (horas)")
plt.ylabel("Temperatura (°C)")
plt.tick_params(labelsize = 8)
plt.show()
```



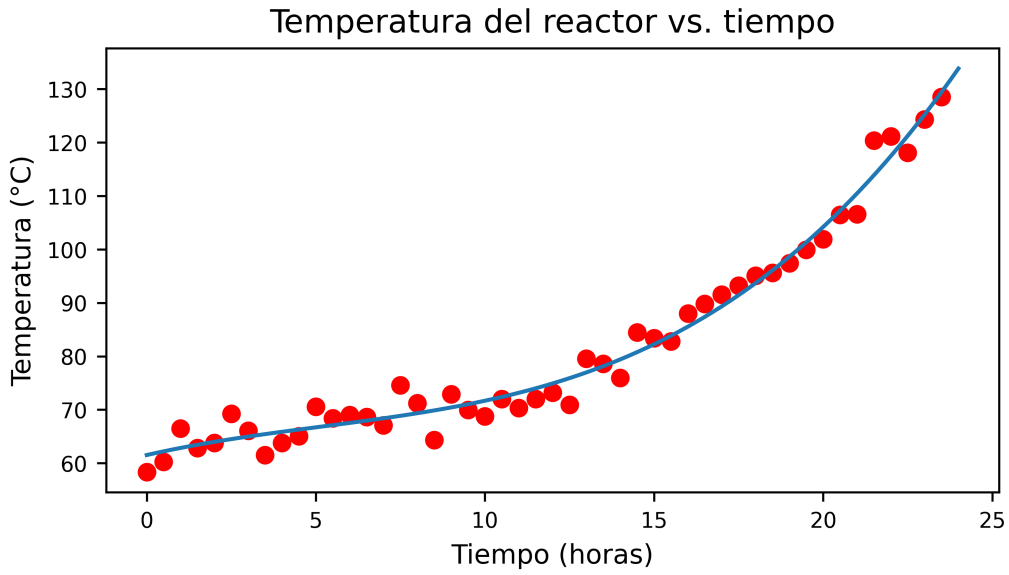
[12]:
```
# ------------------------------------------------------------
# Encontrar el mínimo de la función:
# f(beta) = sum((y - p(x)) ** 2)
# p(x) = beta_0 + beta_1 * x + beta_2 * x ** 2 + beta_3 * x ** 3
# ------------------------------------------------------------

# Definimos la función objetivo, gradiente y Hessiana
def f_objetivo(beta, x, y):
    e = y - beta[0] - beta[1] * x - beta[2] * x ** 2 - beta[3] * x ** 3
    return np.sum(e ** 2)

def grad_f(beta, x, y):
    e = y - beta[0] - beta[1] * x - beta[2] * x ** 2 - beta[3] * x ** 3
    d0 = -2 * np.sum(e)
    d1 = -2 * np.sum(e * x)
    d2 = -2 * np.sum(e * x ** 2)
    d3 = -2 * np.sum(e * x ** 3)
    return np.array([d0, d1, d2, d3])

def hess_f(beta, x, y):
    e = y - beta[0] - beta[1] * x - beta[2] * x ** 2 - beta[3] * x ** 3
    n = len(x)
    d00 = 2 * n
    d01 = d10 = 2 * np.sum(x)
    d02 = d20 = 2 * np.sum(x ** 2)
    d03 = d30 = 2 * np.sum(x ** 3)
    d11 = 2 * np.sum(x ** 2)
```

```
    d12 = d21 = 2 * np.sum(x ** 3)
    d13 = d31 = 2 * np.sum(x ** 4)
    d22 = 2 * np.sum(x ** 4)
    d23 = d32 = 2 * np.sum(x ** 5)
    d33 = 2 * np.sum(x ** 6)
    return np.array([[d00, d01, d02, d03],
                     [d10, d11, d12, d13],
                     [d20, d21, d22, d23],
                     [d30, d31, d32, d33]])

# Definición del punto inicial
max_iter = 10000
tol = 1e-6
beta_actual = np.array([60, 2, 0, 0])

# Algoritmo de Newton
beta_historial = np.array([beta_actual])
for i in range(max_iter):
    beta_nuevo = beta_actual - np.linalg.inv(hess_f(beta_actual, x, y)) @␣
 ↪grad_f(beta_actual, x, y)
    beta_historial = np.vstack((beta_historial, beta_nuevo))
    criterio_1 = np.linalg.norm(grad_f(beta_nuevo, x, y))
    criterio_2 = np.linalg.norm(beta_nuevo - beta_actual)
    if (criterio_1 < tol or criterio_2 < tol):
        break
    beta_actual = beta_nuevo

print("Beta estimado =", beta_nuevo)
print("Iteraciones =", len(beta_historial))
```

```
Beta estimado = [ 6.15202186e+01  1.44366927e+00 -1.19963154e-01
7.72391985e-03]
Iteraciones = 2
```

[13]:
```
# Diagrama de dispersión
plt.figure(figsize = (6, 3), dpi = resolucion)
plt.plot(x, y, "ro")
plt.title("Temperatura del reactor vs. tiempo")
plt.xlabel("Tiempo (horas)")
plt.ylabel("Temperatura (°C)")
plt.tick_params(labelsize = 8)

# Ajuste
x_ajuste = np.linspace(0, 24, 100)
y_ajuste = beta_nuevo[0] + beta_nuevo[1] * x_ajuste + beta_nuevo[2] * x_ajuste␣
 ↪** 2 + beta_nuevo[3] * x_ajuste ** 3
plt.plot(x_ajuste, y_ajuste)
```

```
plt.show()
```



2 Método Quasi-Newton

2.1 La idea fundamental

El método Quasi-Newton busca encontrar el mínimo de una función de manera eficiente. Pertenece a la familia de los métodos de "segundo orden" porque, igual que el método de Newton, utiliza la información sobre la curvatura de la función para acelerar la convergencia. Sin embargo, su principal ventaja es que **evita el cálculo directo, el almacenamiento y la inversión de la matriz Hessiana**.

En esencia, los métodos Quasi-Newton son un punto intermedio entre el lento **método de descenso de gradiente** y el computacionalmente exigente **método de Newton**.

Recordemos la fórmula de actualización del método de Newton:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{H}_f^{-1}(\mathbf{x}_k)\nabla f(\mathbf{x}_k)$$

El principal cuello de botella aquí es el término $\mathbf{H}_f^{-1}(\mathbf{x}_k)$, la inversa de la matriz Hessiana. Calcular la Hessiana y luego invertirla es una tarea muy compleja, especialmente para funciones con muchas variables, comunes en estadística y machine learning. La solución que proponen los **métodos Quasi-Newton** es **no calcular la Hessiana explícitamente**. En su lugar, **construir una aproximación de su inversa**, denotada como $\mathbf{H}_k$, que se refina en cada iteración.

La fórmula de actualización se transforma en:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{H}_k \nabla f(\mathbf{x}_k)$$

Donde:

- $\mathbf{H}_k$ es nuestra aproximación a la inversa de la Hessiana en la iteración k.
- α_k es una tasa de aprendizaje que se determina mediante una búsqueda asegurar un descenso adecuado.

2.2 Algoritmo Quasi-Newton

El procedimiento de un método Quasi-Newton, se puede resumir en los siguientes pasos:

1. **Inicialización:**
   - Se elige un punto de partida $\mathbf{x}_0$.
   - Se inicializa la aproximación de la inversa de la Hessiana, $\mathbf{H}_0$. Una elección común y sencilla es la **matriz identidad** ($\mathbf{I}$):

$$\mathbf{H}_0 = \mathbf{I}$$

2. **Iteración:** Para $k = 0, 1, 2, ...$, se calcula la dirección de búsqueda multiplicando la matriz actual $\mathbf{H}_k$ por el gradiente:

$$\mathbf{p}_k = -\mathbf{H}_k \nabla f(\mathbf{x}_k)$$

3. **Búsqueda de la tasa de aprendizaje:** Se encuentra un α_k adecuado en la dirección $\mathbf{p}_k$.
4. **Actualizar la posición:** Se calcula el siguiente punto de la secuencia:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

5. **Verificar la convergencia:** A través de los criterios de la magnitud del gradiente o cambio de posición, se evalua la convergencia en el nuevo punto.
6. **Actualizar la aproximación:** Se actualiza $\mathbf{H}_k$ a $\mathbf{H}_{k+1}$ mediante una fórmula específica. Esta actualización es una serie de operaciones matriciales simples (de orden $O(n^2)$), mucho más rápidas que la inversión ($O(n^3)$).

Este ciclo se repite hasta que se alcanza la convergencia. La matriz $\mathbf{H}_k$ acumula gradualmente información sobre la curvatura de la función, permitiendo que los pasos sean cada vez más precisos y directos hacia el mínimo.

2.3 Actualización BFGS

El objetivo es construir una nueva matriz $\mathbf{H}_{k+1}$ (nuestra aproximación a la inversa de la Hessiana) a partir de la matriz anterior $\mathbf{H}_k$. Esta nueva matriz debe incorporar la información de la curvatura que acabamos de aprender al movernos del punto $\mathbf{x}_k$ al $\mathbf{x}_{k+1}$. La fórmula es la siguiente:

$$\mathbf{H}_{k+1} = \left(\mathbf{I} - \frac{\mathbf{s}_k \mathbf{y}_k^\top}{\mathbf{y}_k^\top \mathbf{s}_k}\right) \mathbf{H}_k \left(\mathbf{I} - \frac{\mathbf{y}_k \mathbf{s}_k^\top}{\mathbf{y}_k^\top \mathbf{s}_k}\right) + \frac{\mathbf{s}_k \mathbf{s}_k^\top}{\mathbf{y}_k^\top \mathbf{s}_k}$$

Donde:

- $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$, el vector del último paso que dimos.
- $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$, el vector del cambio observado en el gradiente.
- $\mathbf{H}_k$ es nuestra matriz de aproximación actual.
- $\mathbf{I}$ es la matriz identidad.

2.4 Búsqueda de la tasa de aprendizaje

La búsqueda de la tasa de aprendizaje es un componente crucial en casi todos los algoritmos de optimización, incluido el método Quasi-Newton. Su objetivo es responder a una pregunta aparentemente simple: "Ya que hemos elegido una dirección de descenso ($\mathbf{p}_k$), ¿qué tan largo debe ser el paso (α_k) que damos en esa dirección?". Una tasa de aprendizaje demasiada larga podría "saltarse" el mínimo, y una tasa de aprendizaje demasiada corta haría la convergencia innecesariamente lenta.

Una vez que hemos calculado la dirección de descenso $\mathbf{p}_k$ a partir de nuestro punto actual $\mathbf{x}_k$, la actualización será:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

El método más común y sencillo para encontrar un $\alpha_k > 0$ adecuado es la **búsqueda por retroceso (backtracking)**:

1. **Inicialización:** Se elige un valor para α_k, generalmente $\alpha_k = 1$.
2. **Verificar la condición:** Se comprueba si α_k produce una "disminución suficiente" en el valor de la función. La condición más común es la **Condición de Armijo**.
3. **Reducir el paso si es necesario:** Si la condición no se cumple, el paso es demasiado largo. Se reduce α_k multiplicándolo por un factor de retroceso ρ (un valor típico es $\rho = 0.5$).
4. **Repetir:** Se vuelve al paso 2 con el nuevo y más corto α_k.

La **Condición de Armijo** se formaliza de la siguiente manera:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c\alpha_k \nabla f(\mathbf{x}_k)^\top \mathbf{p}_k$$

Donde:

- $f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$: Es el valor de la función en el punto de llegada propuesto.
- $f(\mathbf{x}_k)$: Es el valor actual de la función.
- $\nabla f(\mathbf{x}_k)^\top \mathbf{p}_k$: Es la derivada direccional en la dirección $\mathbf{p}_k$.
- c: Es una constante pequeña, típicamente $c \approx 10^{-4}$.

2.5 Implementación

[14]:
```python
# Aproximación de inversa de la Hessiana
def h_bfgs(h_actual, s_actual, y_actual):
    yts_actual = np.dot(y_actual, s_actual)
    if yts_actual <= 1e-10:
        return h_actual
    n = h_actual.shape[0]
    I = np.identity(n)
    aux1 = I - np.outer(s_actual, y_actual) / yts_actual
    aux2 = I - np.outer(y_actual, s_actual) / yts_actual
    aux3 = np.outer(s_actual, s_actual) / yts_actual
    h_nuevo = aux1 @ h_actual @ aux2 + aux3
    return h_nuevo
```

[15]:
```python
# Encuentra el tamaño de la tasa de aprendizaje
def backtracking(f, grad_f, x_actual, p_actual, alpha_inicial = 1, rho = 0.5, c
 ↪= 1e-4):
    alpha = alpha_inicial
    gradiente_actual = grad_f(x_actual)
    while f(x_actual + alpha * p_actual) > f(x_actual) + c * alpha * np.
 ↪dot(gradiente_actual, p_actual):
        alpha = alpha * rho
    return alpha
```

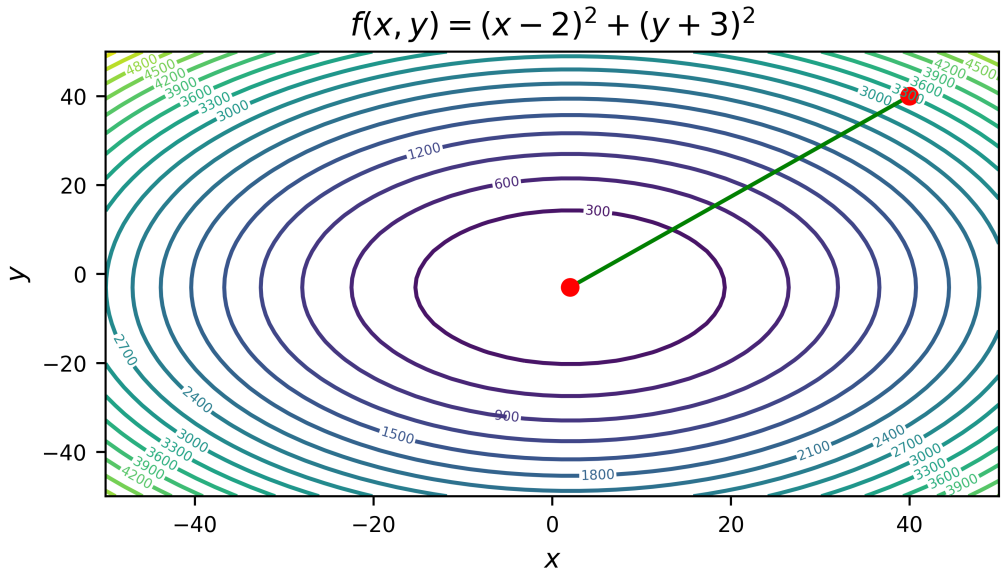
[16]:
```python
# Algoritmo de optimización Quasi-Newton con BFGS
def quasi_newton(f, grad_f, x_actual, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    h_actual = np.identity(len(x_actual))
    for i in range(max_iter):
        # Calcular la dirección de búsqueda
        grad_actual = grad_f(x_actual)
        p_actual = - h_actual @ grad_actual
        # Búsqueda de la tasa de aprendizaje
        alpha_actual = backtracking(f, grad_f, x_actual, p_actual)
        # Actualizar la posición
        x_nuevo = x_actual + alpha_actual * p_actual
        x_historial = np.vstack((x_historial, x_nuevo))
        # Verificar la convergencia
        grad_nuevo = grad_f(x_nuevo)
        criterio1 = np.linalg.norm(grad_nuevo)
        criterio2 = np.linalg.norm(x_nuevo - x_actual)
        if criterio1 < tol or criterio2 < tol:
            break
        # Calcular aproximar la matriz H_k
        s_actual = x_nuevo - x_actual
        y_actual = grad_nuevo - grad_actual
```

```
        h_actual = h_bfgs(h_actual, s_actual, y_actual)
        x_actual = x_nuevo
    return x_nuevo, x_historial
```

[17]:
```
# ------------------------------------
# Encontrar el mínimo de la función:
# f(x, y) = (x - 2) ** 2 + (y + 3) ** 2
# ------------------------------------

# Definición del punto inicial
x_actual = np.array([40, 40])

# Ejecución del algoritmo y resultados
resultado = quasi_newton(paraboloide, grad_paraboloide, x_actual)

print("(x, y) =", resultado[0])
print("f(x, y) =", paraboloide(resultado[0]))
print("Iteraciones =", len(resultado[1]))
```

```
(x, y) = [ 2. -3.]
f(x, y) = 0.0
Iteraciones = 2
```

[18]:
```
# Graficar la función
x = np.linspace(-50, 50, 50)
y = np.linspace(-50, 50, 50)
X, Y = np.meshgrid(x, y)
Z = paraboloide([X, Y])

plt.figure(figsize = (6, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")
plt.clabel(contour, inline = True, fontsize = 5)
plt.title("$f(x, y) = (x - 2)^2 + (y + 3)^2$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.tick_params(labelsize = 8)

# Puntos de trayectoria
plt.plot(resultado[1][:, 0], resultado[1][:, 1], "g-")
plt.plot(resultado[1][:, 0], resultado[1][:, 1], "ro")
plt.show()
```

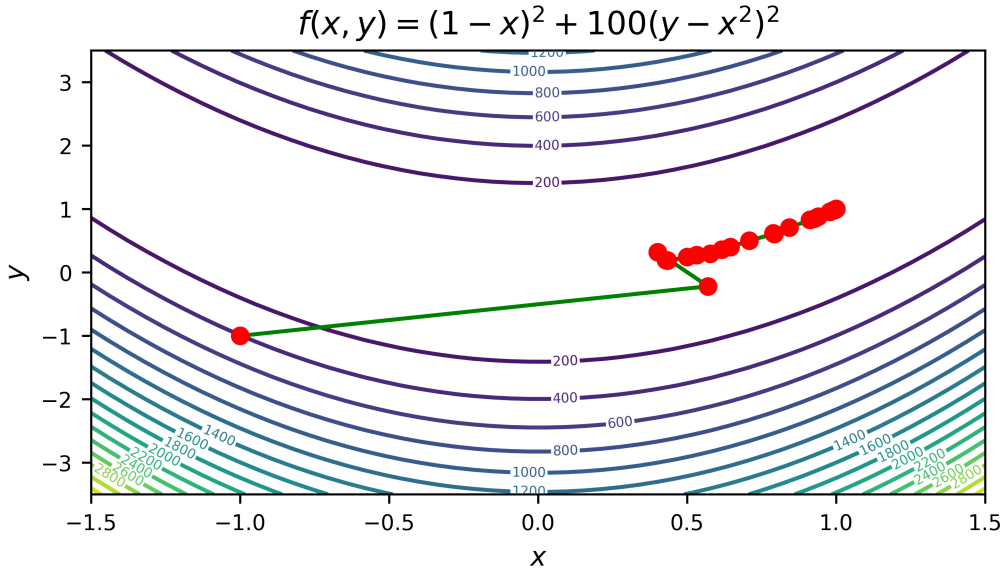


[19]:
```
# ------------------------------------------------
# Encontrar el mínimo de la función de Rosenbrock:
# f(x, y) = (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
# ------------------------------------------------

# Definición del punto inicial
x_actual = np.array([-1, -1])

# Ejecución del algoritmo y resultados
resultado = quasi_newton(rosenbrock, grad_rosenbrock, x_actual)

print("(x, y) =", resultado[0])
print("f(x, y) =", rosenbrock(resultado[0]))
print("Iteraciones =", len(resultado[1]))
```

```
(x, y) = [0.99999999 0.99999997]
f(x, y) = 2.441076501315356e-16
Iteraciones = 25
```

[20]:
```
# Graficar la función
x = np.linspace(-1.5, 1.5, 50)
y = np.linspace(-3.5, 3.5, 50)
X, Y = np.meshgrid(x, y)
Z = rosenbrock([X, Y])

plt.figure(figsize = (6, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")
plt.clabel(contour, inline = True, fontsize = 5)
```

```
plt.title("$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.tick_params(labelsize = 8)

# Puntos de trayectoria
plt.plot(resultado[1][:, 0], resultado[1][:, 1], "g-")
plt.plot(resultado[1][:, 0], resultado[1][:, 1], "ro")
plt.show()
```



Juan F. Olivares Pacheco (jfolivar@uda.cl)

Universidad de Atacama, Facultad de Ingeniería, Departamento de Matemática