403 MÉTODO DE NEWTON

September 25, 2025

1 Aproximación lineal 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.

2 Formulación iterativa 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.

3 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, $f'(x)$.
- **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**:

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

4 El algoritmo de Newton para optimización

El método de Newton es un poderoso algoritmo iterativo diseñado para encontrar el punto que minimiza o maximiza una función. Para lograrlo, se basa en el análisis de las derivadas de la función en un punto para decidir la dirección hacia un mejor punto en el siguiente paso.

Para entender el método, 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 no encuentra la solución en un solo paso. En su lugar, 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.

5 Implementación

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
resolucion = 600
```

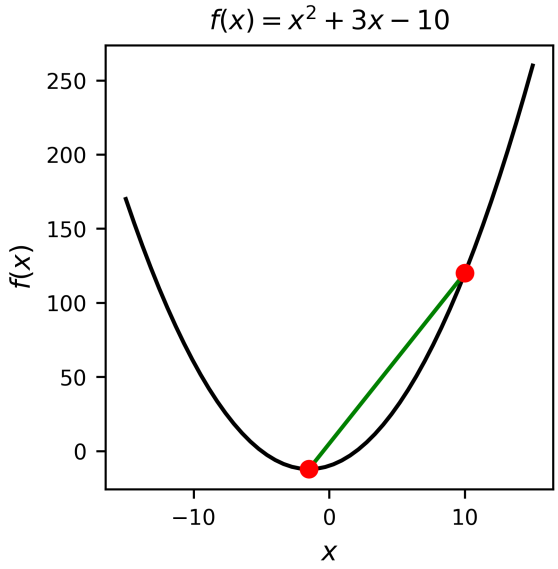
[3]:
```
# ---------------------------------
# Encontrar el mínimo de la función:
# f(x) = x ** 2 + 3 * x - 10
# ---------------------------------

# Definición de la función objetivo
def f_uni(x):
    return x ** 2 + 3 * x - 10

def grad_f_uni(x):
```

```
    return 2 * x + 3

def hess_f_uni(x):
    return 2

# Algoritmo de Newton
def newton_uni(x_actual, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - grad_f_uni(x_actual) / hess_f_uni(x_actual)
        x_historial = np.append(x_historial, x_nuevo)
        criterio_1 = np.linalg.norm(grad_f_uni(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial

# Definición del punto inicial
x_actual = 10

# Ejecución del algoritmo y resultados
r_uni = newton_uni(x_actual)

print("x =", r_uni[0])
print("f(x) =", f_uni(r_uni[0]))
print("Iteraciones =", len(r_uni[1]))
```

```
x = -1.5
f(x) = -12.25
Iteraciones = 2
```

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

plt.figure(figsize = (3, 3), dpi = resolucion)
plt.plot(x, y, color = "black", zorder = 1)

plt.title("$f(x) = x^2 + 3 x - 10$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$f(x)$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_uni[1], f_uni(r_uni[1]), color = "green", zorder = 2)
plt.scatter(r_uni[1], f_uni(r_uni[1]), color = "red", zorder = 3)
plt.tick_params(labelsize = 8)
```

```
plt.show()
```



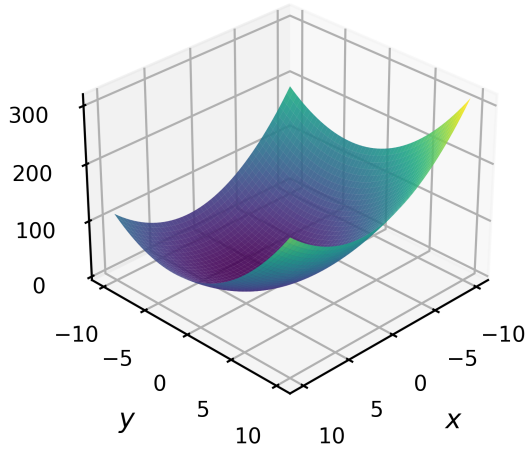
[5]:
```
# ------------------------------------
# Encontrar el mínimo de la función:
# f(x, y) = (x - 2) ** 2 + (y + 3) ** 2
# ------------------------------------

# Definición de la función objetivo
def f_multi(x):
    return (x[0] - 2) ** 2 + (x[1] + 3) ** 2

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

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

# Algoritmo de Newton
def newton_multi(x_actual, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - np.linalg.inv(hess_f_multi(x_actual)) @␣
 ↪grad_f_multi(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_f_multi(x_nuevo))
```

```
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial

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

# Ejecución del algoritmo y resultados
r_multi = newton_multi(x_actual)

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

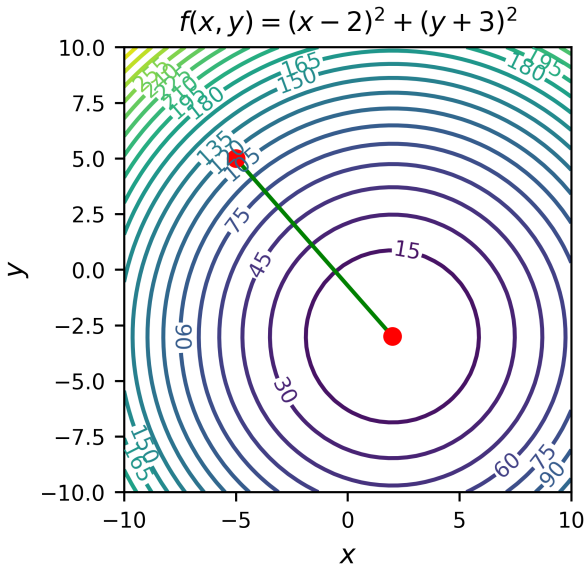
```
(x, y) = [ 2. -3.]
f(x, y) = 0.0
Iteraciones = 2
```

[6]:
```
# Graficar la función
x = np.linspace(-10, 10, 50)
y = np.linspace(-10, 10, 50)
X, Y = np.meshgrid(x, y)
Z = f_multi([X, Y])

fig = plt.figure(figsize = (3, 3), dpi = resolucion)
ax = fig.add_subplot(111, projection = "3d")
ax.plot_surface(X, Y, Z, cmap = "viridis", alpha = 0.9)
ax.view_init(elev = 30, azim = 45)

ax.set_title("$f(x, y) = (x - 2)^2 + (y + 3)^2$", fontsize = 10)
ax.set_xlabel("$x$", fontsize = 10)
ax.set_ylabel("$y$", fontsize = 10)
ax.tick_params(labelsize = 8)
plt.show()
```

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



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

plt.figure(figsize = (3, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")

plt.clabel(contour, inline = 1, fontsize = 8)
plt.title("$f(x, y) = (x - 2)^2 + (y + 3)^2$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$y$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_multi[1][:, 0], r_multi[1][:, 1], color = "green", zorder = 2)
plt.scatter(r_multi[1][:, 0], r_multi[1][:, 1], color = "red", zorder = 3)
plt.tick_params(labelsize = 8)
plt.show()
```

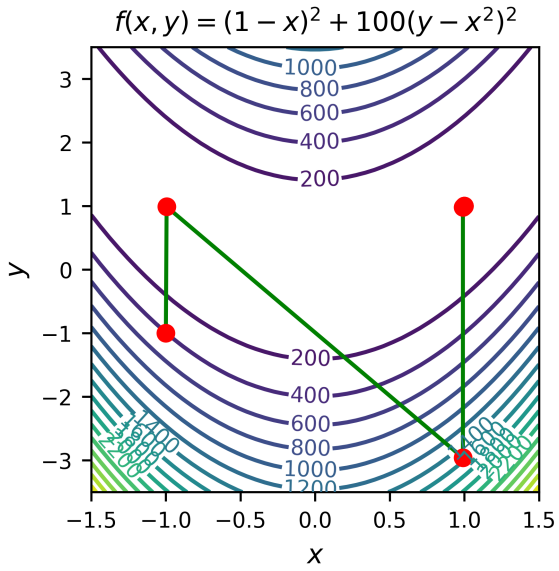


[8]:
```python
# ------------------------------------------------
# Encontrar el mínimo de la función de Rosenbrock:
# f(x, y) = (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
# ------------------------------------------------

# Definimos la función objetivo
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]])

# Algoritmo de Newton
def newton_rosenbrock(x_actual, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        x_nuevo = x_actual - np.linalg.inv(hess_rosenbrock(x_actual)) @␣
↪grad_rosenbrock(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
```

```
        criterio_1 = np.linalg.norm(grad_rosenbrock(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return (x_nuevo, x_historial)

# Definición del punto inicial
x_actual = np.array([-1.0, -1.0])

# Ejecución del algoritmo y resultados
r_rosenbrock = newton_rosenbrock(x_actual)

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

```
x = [1. 1.]
f(x) = 3.4781872520856105e-23
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 = (3, 3), dpi = resolucion)
contour = plt.contour(X, Y, Z, levels = 20, cmap = "viridis")

plt.clabel(contour, inline = 1, fontsize = 8)
plt.title("$f(x, y) = (1 - x)^2 + 100 (y - x^2)^2$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$y$", fontsize = 10)

# Puntos de trayectoria
plt.plot(r_rosenbrock[1][:, 0], r_rosenbrock[1][:, 1], color = "green", zorder␣
 ↪= 2)
plt.scatter(r_rosenbrock[1][:, 0], r_rosenbrock[1][:, 1], color = "red", zorder␣
 ↪= 3)
plt.tick_params(labelsize = 8)
plt.show()
```



6 Consideraciones prácticas

Para implementar el método de Newton de manera efectiva y robusta en problemas reales, es fundamental tener en cuenta las consideraciones prácticas. Además de establecer criterios claros 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
# ------------------------------------------------
# Encontrar el mínimo de la función de Rosenbrock:
# f(x, y) = (1 - x) ** 2 + 100 * (y - x ** 2) ** 2
# ------------------------------------------------

# Definimos la función objetivo
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])
```

```python
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]])

# Algoritmo de Newton
def newton_rosenbrock(x_actual, max_iter = 10000, tol = 1e-6):
    x_historial = np.array([x_actual])
    for i in range(max_iter):
        hessian = hess_rosenbrock(x_actual)
        eigenvalues = np.linalg.eigvals(hessian)
        if np.any(eigenvalues <= 0):
            hessian = hessian + (np.abs(np.min(eigenvalues)) + 0.1) * np.
 ↪identity(2)
        x_nuevo = x_actual - np.linalg.inv(hessian) @ grad_rosenbrock(x_actual)
        x_historial = np.vstack((x_historial, x_nuevo))
        criterio_1 = np.linalg.norm(grad_rosenbrock(x_nuevo))
        criterio_2 = np.linalg.norm(x_nuevo - x_actual)
        if (criterio_1 < tol or criterio_2 < tol):
            break
        x_actual = x_nuevo
    return x_nuevo, x_historial

# Definición del punto inicial
x_actual = np.array([-1.5, 7.5])

# Ejecución del algoritmo y resultados
r_rosenbrock = newton_rosenbrock(x_actual)

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

```
x = [1. 1.]
f(x) = 3.9377613155056583e-20
Iteraciones = 8
```

7 Aplicaciones

7.1 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:

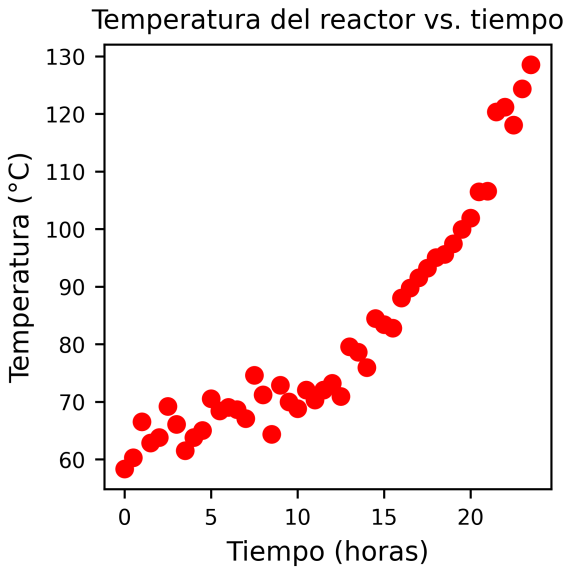
- Interpolar valores en momentos no medidos

- 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 = (3, 3), dpi = resolucion)
plt.scatter(x, y, color = "red")
plt.title("Temperatura del reactor vs. tiempo", fontsize = 10)
plt.xlabel("Tiempo (horas)", fontsize = 10)
plt.ylabel("Temperatura (°C)", fontsize = 10)
plt.tick_params(labelsize = 8)
plt.show()
```



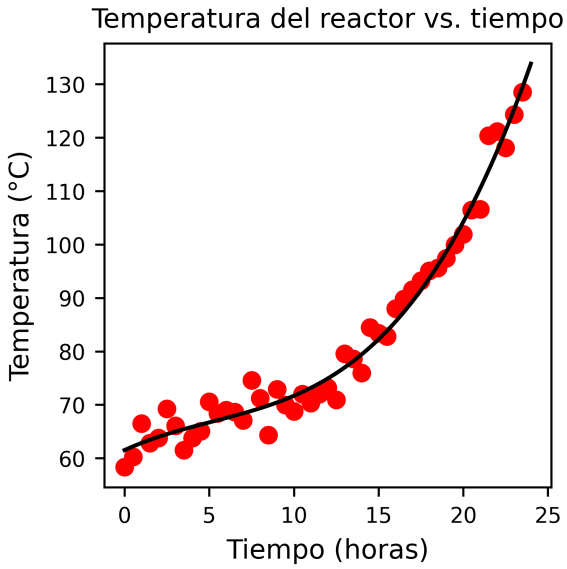
[12]:
```python
# ------------------------------------------------------------
# Encontrar el mínimo de la función:
# f(beta) = sum((y - p(x)) ** 2)
# p(x) = beta_0 + beta_1 * x + beta_2 * x ** 2 + beta_3 * x ** 3
# ------------------------------------------------------------

# Definimos la función objetivo
# -----------------------------
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]])

# Algoritmo de Newton
def newton(beta_actual, x, y, max_iter = 10000, tol = 1e-6):
    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
    return beta_nuevo, beta_historial

# Definición del punto inicial
beta_actual = np.array([60, 2, 0, 0])

# Ejecución del algoritmo y resultados
r = newton(beta_actual, x, y)
beta_estimado = r[0]

print("Beta estimado =", beta_estimado)
print("Iteraciones =", len(r[1]))
```

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

[13]:
```
# Diagrama de dispersión
plt.figure(figsize = (3, 3), dpi = resolucion)
plt.scatter(x, y, color = "red")
plt.title("Temperatura del reactor vs. tiempo", fontsize = 10)
plt.xlabel("Tiempo (horas)", fontsize = 10)
plt.ylabel("Temperatura (°C)", fontsize = 10)

# Ajuste
```

```
x_ajuste = np.linspace(0, 24, 100)
y_ajuste = beta_estimado[0] + beta_estimado[1] * x_ajuste + beta_estimado[2] *␣
 ↪x_ajuste ** 2 + beta_estimado[3] * x_ajuste ** 3
plt.plot(x_ajuste, y_ajuste, color = "black")
plt.tick_params(labelsize = 8)
plt.show()
```



8 Ejercicios

Minimice la función $f(x) = (x-2)^3$.

Minimice la función $f(x, y) = \sin(x) + \cos(y)$.

Minimice la función de Beale $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$.

Minimice la función Himmelblau $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$.

Minimice la función $f(x, y, z) = x^2 + y^2 + z^2$.

Minimice la función $f(x, y, z) = (x-1)^2 + (y-2)^2 + (z-3)^2 + \sin(x)\cos(y)\exp(z)$.