C6 MODELO PREDICTIVO DE LA TEMPERATURA PARA UN REACTOR QUÍMICO

December 1, 2025

1 Objetivo del proyecto

En ingeniería química, monitorear y predecir la temperatura de un reactor es vital para garantizar la seguridad, le eficiencia y la calidad del producto. Las fluctuaciones pueden indicar problemas o ser parte normal del proceso.

El objetivo es modelar la tendencia temporal de la temperatura de un reactor durante las primeras 24 horas de un proceso de síntesis. Se busca ajustar un **polinomio de grado 3** a los datos de los sensores:

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

Para realizar el ajuste, se implementarán y compararán tres métodos de optimización para encontrar los parámetros del modelo que minimicen el error cuadrático:

- Método de descenso de gradiente
- Método de Newton
- Método Quasi-Newton (BFGS)

Un modelo preciso permite: (i) Interpolar valores en momentos no medidos, (ii) predecir el comportamiento a corto plazo, y (iii) identificar patrones anómalos.

2 Datos del problema

Los datos consisten en mediciones de temperatura tomadas cada media hora.

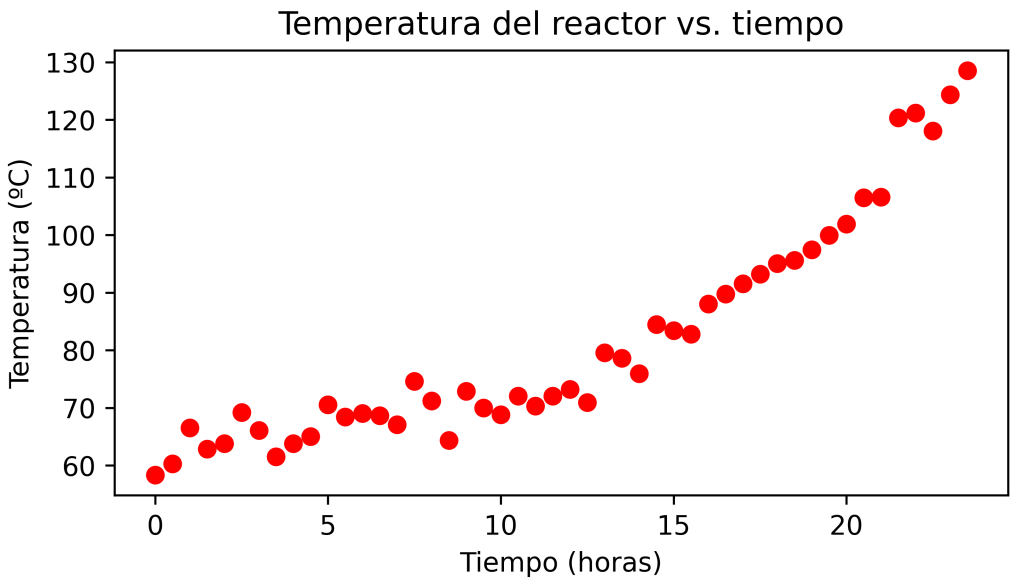
- **Variable X:** Tiempo en horas.
- **Variable Y:** Temperatura en ºC.

[1]:
```
# Cargar librerías necesarias
# ---------------------------
import numpy as np
import matplotlib.pyplot as plt

# Configuraciones inciales
# ------------------------
```

```
plt.rcParams["figure.dpi"] = 600
```

[2]:
```
# 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])
```

[3]:
```
# Diagrama de dispersión
# ----------------------
fig, ax = plt.subplots(figsize = (6, 3))
ax.plot(x, y, "ro")
ax.set_title("Temperatura del reactor vs. tiempo")
ax.set_xlabel("Tiempo (horas)")
ax.set_ylabel("Temperatura (ºC)")
plt.show()
```



El diagrama de dispersión de los datos, muestra una clara tendencia no lineal ascendente, que un simple modelo lineal ($y = \beta_0 + \beta_1 x$) no podría capturar adecuadamente.

3 Formulación del problema de optimización

Se busca encontrar el vector de parámetros $\beta = [\beta_0, \beta_1, \beta_2, \beta_3]$ que define el polinomio:

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

El objetivo es minimizar la **suma de errores cuadráticos**, es decir, la diferencia entre las temperaturas reales (y_i) y las temperaturas predichas por el polinomio ($p(x_i)$).

$$f(\beta) = \sum_{i=1}^{n} (y_i - p(x_i))^2 = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3))^2$$

Para aplicar los algoritmos, necesitamos definir su gradiente (primera derivada) y su matriz Hessiana (segunda derivada).

- **Vector gradietene** $\nabla f(\beta)$: Es un vector 4×1 que contine las derivadas parciales de $f(\beta)$ con respecto a cada parámetro β_j. Mide la dirección de máximo ascenso de la función objetivo.

$$\nabla f(\beta) = \begin{bmatrix} \frac{\partial f}{\partial \beta_0} \\ \frac{\partial f}{\partial \beta_1} \\ \frac{\partial f}{\partial \beta_2} \\ \frac{\partial f}{\partial \beta_3} \end{bmatrix} = \begin{bmatrix} -2\sum e_i \\ -2\sum e_i x_i \\ -2\sum e_i x_i^2 \\ -2\sum e_i x_i^3 \end{bmatrix}$$

donde e_i es el error en el punto i, $e_i = y_i - p(x_i)$.

- **Matriz Hessiana $\mathbf{H}_f(\beta)$**: Es una matriz simétrica de 4×4 que contiene las segundas derivadas parciales. Describe la curvatura de la función objetivo.

$$\mathbf{H}_f(\beta) = \begin{bmatrix} \frac{\partial^2 f}{\partial \beta_0^2} & \frac{\partial^2 f}{\partial \beta_0 \partial \beta_1} & \frac{\partial^2 f}{\partial \beta_0 \partial \beta_2} & \frac{\partial^2 f}{\partial \beta_0 \partial \beta_3} \\ \frac{\partial^2 f}{\partial \beta_1 \partial \beta_0} & \frac{\partial^2 f}{\partial \beta_1^2} & \frac{\partial^2 f}{\partial \beta_1 \partial \beta_2} & \frac{\partial^2 f}{\partial \beta_1 \partial \beta_3} \\ \frac{\partial^2 f}{\partial \beta_2 \partial \beta_0} & \frac{\partial^2 f}{\partial \beta_2 \partial \beta_1} & \frac{\partial^2 f}{\partial \beta_2^2} & \frac{\partial^2 f}{\partial \beta_2 \partial \beta_3} \\ \frac{\partial^2 f}{\partial \beta_3 \partial \beta_0} & \frac{\partial^2 f}{\partial \beta_3 \partial \beta_1} & \frac{\partial^2 f}{\partial \beta_3 \partial \beta_2} & \frac{\partial^2 f}{\partial \beta_3^2} \end{bmatrix} = \begin{bmatrix} 2n & 2\sum x_i & 2\sum x_i^2 & 2\sum x_i^3 \\ 2\sum x_i & 2\sum x_i^2 & 2\sum x_i^3 & 2\sum x_i^4 \\ 2\sum x_i^2 & 2\sum x_i^3 & 2\sum x_i^4 & 2\sum x_i^5 \\ 2\sum x_i^3 & 2\sum x_i^4 & 2\sum x_i^5 & 2\sum x_i^6 \end{bmatrix}$$

Notemos que, ninguno de los términos de la matriz Hessiana depende de los parámetros β. Solo dependen de los datos de entrada x. Esto significa que la **Hessiana es constante**. La curvatura de la función de error es la misma en cualquier punto del espacio de parámetros.

[4]:
```python
# ----------------------------
# Función objetivo a minimizar
# ----------------------------
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)

# --------------------------------
# Gradiente de la función objetivo
# --------------------------------
def g_objetivo(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])

# -------------------------------
# Hessiana de la función objetivo
# -------------------------------
def h_objetivo(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]])
```

4 Implementación y resultados de los métodos de optimización

4.1 Método 1: Descenso de gradiente

La regla de actualización es:

$$\beta_{k+1} = \beta_k - \alpha \nabla f(\beta_k)$$

[5]:
```
# ---------------------------------
# Algoritmo de descenso de gradiente
# ---------------------------------
def dg_rp(beta_actual, x, y, alpha, max_iter = 100000, tol = 1e-6):
    beta_historial = [beta_actual]
    for _ in range(max_iter):
        beta_nuevo = beta_actual - alpha * g_objetivo(beta_actual, x, y)
        beta_historial.append(beta_nuevo)
        criterio_1 = np.linalg.norm(beta_nuevo - beta_actual)
        criterio_2 = np.linalg.norm(g_objetivo(beta_nuevo, x, y))
        if criterio_1 < tol or criterio_2 < tol:
            break
        beta_actual = beta_nuevo
```

```
    beta_historial = np.array(beta_historial)
    return beta_nuevo, beta_historial

# Parámetros y ejecución
# alpha_dg = 0.001
# beta_inicial = np.array([60, 2, 0, 0])
# beta_dg, hist_dg = dg_rp(beta_inicial, x, y, alpha_dg)

# print("--- Descenso de gradiente ---")
# print("Beta estimado =", beta_dg)
# print("Número de iteraciones =", len(hist_dg))
```

4.2 Método 2: Método de Newton

La regla de actualización es:

$$\beta_{k+1} = \beta_k - \mathbf{H}_k^{-1}(\beta_k)\nabla f(\beta_k)$$

[6]:
```
# -------------------
# Algoritmo de Newton
# -------------------
def newton_rp(beta_actual, x, y, max_iter = 10000, tol = 1e-6):
    beta_historial = [beta_actual]
    for _ in range(max_iter):
        beta_nuevo = beta_actual - np.linalg.inv(h_objetivo(beta_actual, x, y))␣
 ↪@ g_objetivo(beta_actual, x, y)
        beta_historial.append(beta_nuevo)
        criterio_1 = np.linalg.norm(beta_nuevo - beta_actual)
        criterio_2 = np.linalg.norm(g_objetivo(beta_nuevo, x, y))
        if criterio_1 < tol or criterio_2 < tol:
            break
        beta_actual = beta_nuevo
    beta_historial = np.array(beta_historial)
    return beta_nuevo, beta_historial

# Parámetros y ejecución
beta_inicial = np.array([60, 2, 0, 0])
beta_newton, hist_newton = newton_rp(beta_inicial, x, y)

print("--- Método de Newton ---")
print("Beta estimado =", beta_newton)
print("Número de iteraciones =", len(hist_newton))
```

```
--- Método de Newton ---
Beta estimado = [ 6.15202186e+01  1.44366927e+00 -1.19963154e-01
7.72391985e-03]
Número de iteraciones = 2
```

4.3 Método 3: Quasi-Newton (BFGS)

Este método aproxima la inversa de la Hessiana en lugar de calcularla directamente, logrando un equilibrio en la la velocidad del método de Newton y la simplicidad del descenso de gradiente.

[7]:
```
# ----------------------
# Algoritmo Quasi-Newton
# ----------------------
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 = s_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

def backtracking(f, grad_f, x_actual, p_actual, alpha_inicial = 1.0, rho = 0.5,␣
 ↪c = 1e-4):
    alpha = alpha_inicial
    gradiente_actual = grad_f(x_actual, x, y)
    while f(x_actual + alpha * p_actual, x, y) > f(x_actual, x, y) + c * alpha␣
 ↪* np.dot(gradiente_actual, p_actual):
        alpha = alpha * rho
    return alpha

def qn_rp(f, grad_f, beta_actual, x, y, max_iter = 10000, tol = 1e-6):
    beta_historial = [beta_actual]
    h_actual = np.identity(len(beta_actual))
    for _ in range(max_iter):
        grad_actual = grad_f(beta_actual, x, y)
        p_actual = -h_actual @ grad_actual
        alpha_actual = backtracking(f, grad_f, beta_actual, p_actual)
        beta_nuevo = beta_actual + alpha_actual * p_actual
        beta_historial.append(beta_nuevo)
        criterio_1 = np.linalg.norm(beta_nuevo - beta_actual)
        criterio_2 = np.linalg.norm(g_objetivo(beta_nuevo, x, y))
        if criterio_1 < tol or criterio_2 < tol:
            break
        grad_nuevo = grad_f(beta_nuevo, x, y)
        s_actual = beta_nuevo - beta_actual
        y_actual = grad_nuevo - grad_actual
        h_actual = h_bfgs(h_actual, s_actual, y_actual)
        beta_actual = beta_nuevo
    beta_historial = np.array(beta_historial)
```

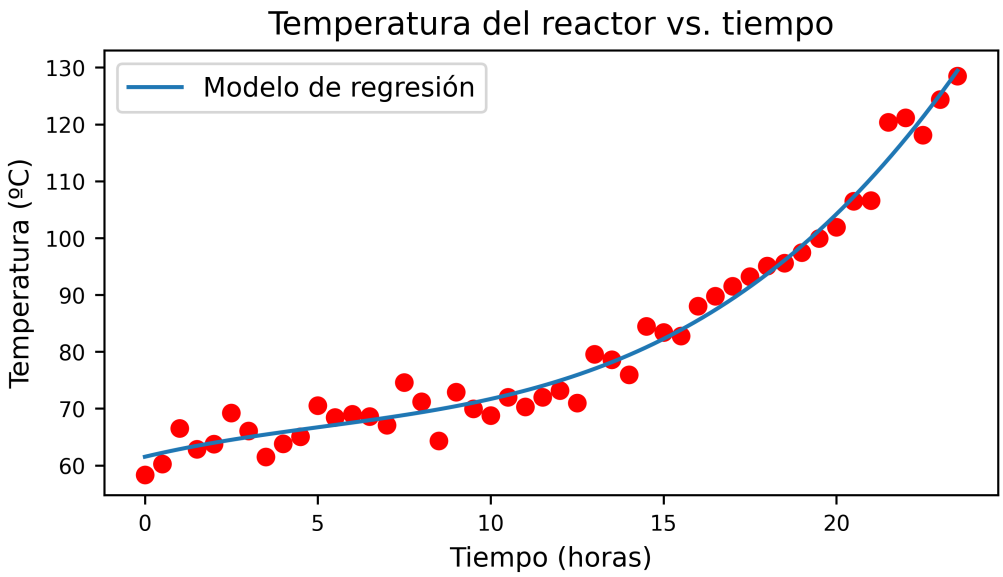
```
    return beta_nuevo, beta_historial

# Ejecución
beta_inicial = np.array([60, 2, 0, 0])
beta_qn, hist_qn = qn_rp(f_objetivo, g_objetivo, beta_inicial, x, y)

print("--- Método de Quasi-Newton ---")
print("Beta estimado =", beta_qn)
print("Número de iteraciones =", len(hist_qn))
```

```
--- Método de Quasi-Newton ---
Beta estimado = [ 6.15202186e+01  1.44366927e+00 -1.19963154e-01
7.72391985e-03]
Número de iteraciones = 11
```

5 Comparación de resultados y conclusión

Una vez ejecutados los tres algoritmos, podemos comparar su rendimiento y el modelo final obtenido.

[8]:
```
# --------------------------------
# Gráfica de la recta de regresión
# --------------------------------
plt.figure(figsize=(6, 3))
plt.plot(x, y, "ro")
x_regresion = np.linspace(min(x), max(x), 100)
y_regresion = beta_newton[0] + beta_newton[1] * x_regresion + beta_newton[2] *␣
 ↪x_regresion ** 2 + beta_newton[3] * x_regresion ** 3
plt.plot(x_regresion, y_regresion, label = "Modelo de regresión")
plt.xlabel("Tiempo (horas)")
plt.ylabel("Temperatura (ºC)")
plt.title("Temperatura del reactor vs. tiempo")
plt.tick_params(labelsize = 8)
plt.legend()
plt.show()
```


Temperatura del reactor vs. tiempo
Modelo de regresión
Temperatura (ºC)
Tiempo (horas)
130
120
110
100
90
80
70
60
0
5
10
15
20