405 SIMULATED ANNEALING

October 17, 2025

1 Introducción

El **Recocido Simulado** es un método de optimización metaheurístico y probabilístico diseñado para encontrar el mínimo (o máximo) global de una función en un espacio de búsqueda grande y complejo. Su principal fortaleza es la capacidad de escapar de mínimos (o máximos) locales, un problema común en muchos otros algoritmos de optimización.

La inspiración del método proviene del proceso de recocido en metalurgia, donde un material se calienta a alta temperatura y luego se enfría lentamente. Este proceso controlado permite que las partículas del material se reorganicen en un estado de menor energía (una estructura cristalina más perfecta), resultando en un material más fuerte y menos frágil.

2 Teoría y conceptos fundamentales

La analogía con la optimización es directa:

- **Estado del sistema:** Corresponde a una solución candidata en el espacio de búsqueda.
- **Energía del estado:** Es el valor de la función objetivo para una solución dada. El objetivo es encontrar el estado de mínima (o máxima) energía (el mínimo o máximo de la función).
- **Temperatura:** Es un parámetro de control del algoritmo, análogo a la temperatura en el proceso físico. Comienza alta y disminuye gradualmente.
- **Enfriamiento:** Es el proceso de reducir la temperatura. La "velocidad" de enfriamiento es crucial para el éxito del algoritmo.

El algoritmo funciona de la siguiente manera:

1. **Inicialización:** Se elige una solución inicial aleatoria S_{actual} y una temperatura inicial alta T_{actual}.
2. **Iteración:** En cada iteración, se genera una nueva solución S_{nueva} en la vecindad de la solución actual.
3. **Evaluación:** Se calcula la diferencia de "energia" (costo) entre la nueva solución y la actual:

$$\Delta E = f(S_{\text{nueva}}) - f(S_{\text{actual}})$$

4. **Decisión:** Si estamos en la búsqueda de un máximo:

- Si la nueva solución es **mejor** ($\Delta E > 0$), se acepta siempre como la nueva solución actual.
- Si la nueva solución es **peor** ($\Delta E \leq 0$), se acepta con una cierta **probabilidad**, dada por $p = e^{-\Delta E/T}$.

Esta aceptación probabilística de soluciones peores es la clave del método. A altas temperaturas, la probabilidad de aceptar un mal movimiento es alta, lo que permite al algoritmo explorar ampliamiente el espacio de búsqueda y "saltar" fuera de los mínimos locales. A medida que la temperatura disminuye, esta probabilidad decrece, y el algoritmo se vuelve más "selectivo", enfocandose en la explotación de las buenas regiones encontradas y convergiendo hacia un mínimo.

5. **Enfriamiento:** Después de un número de iteraciones a una temperatura dada, esta se reduce según un "esquema de enfriamiento". Un esquema común es el enfriamiento geométrico:

$$T_{\text{nueva}} = \alpha \cdot T_{\text{actual}}$$

   donde α es un factor de enfriamiento cercano a 1 (por ejemplo, 0.95).

6. **Finalización:** El algoritmo termina cuando la temperatura alcanza un valor muy bajo (cercano a cero) o cuando no se encuentran mejoras significativas durante un número determinado de iteraciones.

3 Fórmulas matemáticas clave

- **Función objetivo:** $f(\mathbf{x})$, donde $\mathbf{x}$ es el vector de variables.
- **Probabilidad de aceptación**:

$$\mathsf{P}(\text{aceptar } S_{\text{nueva}}) = \begin{cases} 1 & \text{si } f(S_{\text{nueva}}) > f(S_{\text{actual}}) \\ e^{-\Delta E/T} & \text{si } f(S_{\text{nueva}}) \leq f(S_{\text{actual}}) \end{cases}$$

- **Esquema de enfriamiento geométrico**:

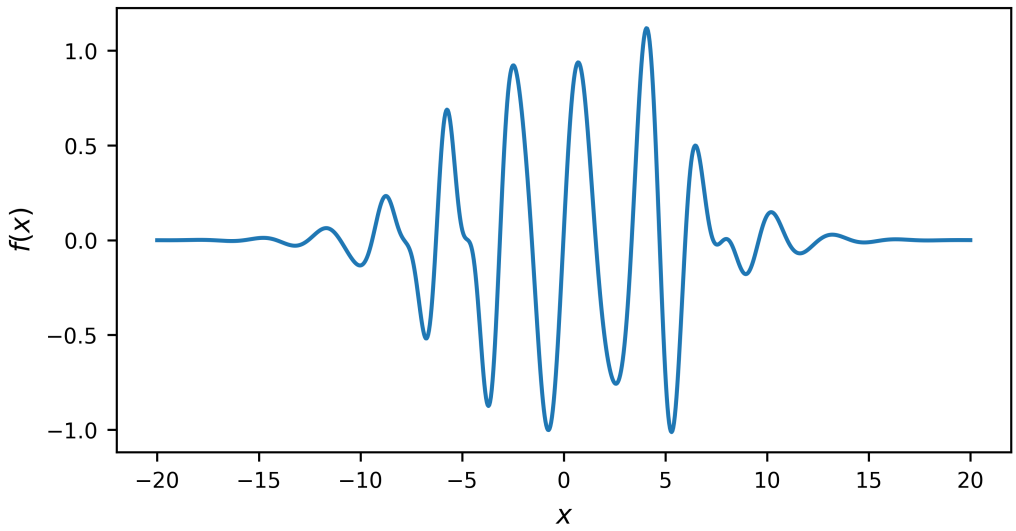
$$T_{k+1} = \alpha \cdot T_k$$

  donde T_k es la temperatura en la iteración k y $0 < \alpha < 1$.

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

[2]:
```
def f_univariada(x):
    aux1 = np.exp(-x ** 2 / 50) * np.sin(2 * x)
    aux2 = 0.5 * np.exp(-(x - 5) ** 2 / 10) * np.cos(3 * x)
    aux3 = 0.3 * np.exp(-(x + 5) ** 2 / 8) * np.sin(4 * x)
    return aux1 + aux2 + aux3
```

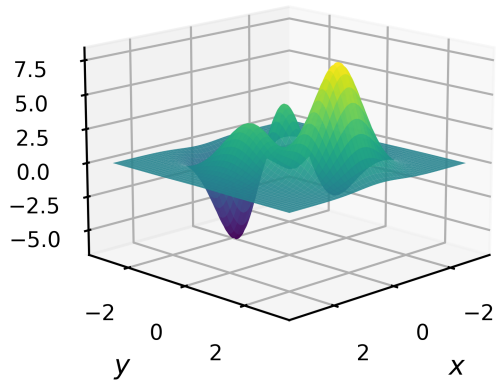
[3]:
```
# Graficar la función
x = np.linspace(-20, 20, 1000)
y = f_univariada(x)

plt.figure(figsize = (6, 3), dpi = resolucion)
plt.plot(x, y)
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.tick_params(labelsize = 8)
plt.show()
```



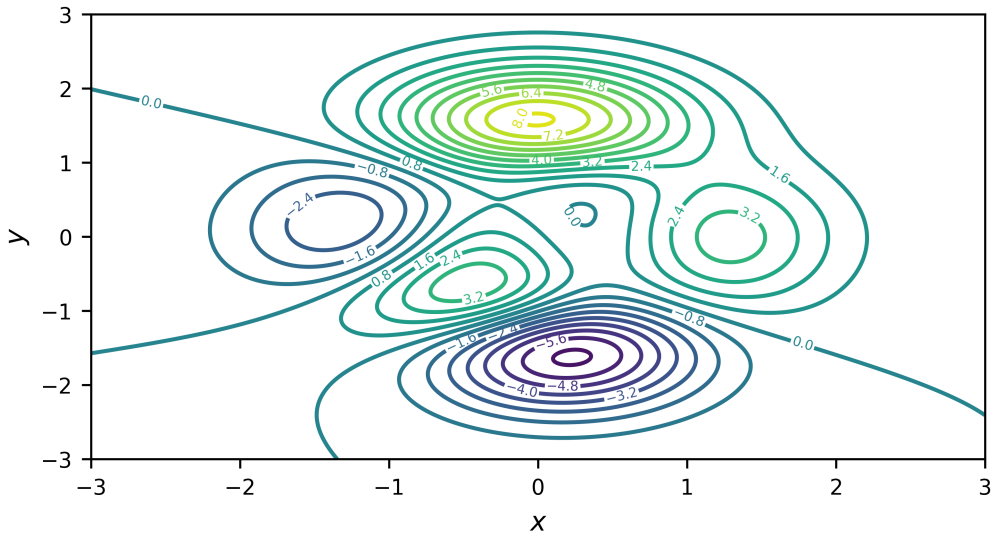
[4]:
```
def f_multivariada(x):
    aux1 = 3 * (1 - x[0]) ** 2 * np.exp(-x[0] ** 2 - (x[1] + 1) ** 2)
    aux2 = -10 * (x[0] / 5 - x[0] ** 3 - x[1] ** 5) * np.exp(-x[0] ** 2 - x[1]␣
 ↪** 2)
    aux3 = -(1 / 3) * np.exp(-(x[0] + 1) ** 2 - x[1] ** 2)
    return aux1 + aux2 + aux3
```

[5]:
```
x = np.linspace(-3, 3, 1000)
y = np.linspace(-3, 3, 1000)
X, Y = np.meshgrid(x, y)
Z = f_multivariada([X, Y])

fig = plt.figure(figsize = (6, 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 = 15, azim = 45)
ax.set_xlabel("$x$")
```

```
ax.set_ylabel("$y$")
ax.tick_params(labelsize = 8)
plt.show()
```



[6]:
```
x = np.linspace(-3, 3, 1000)
y = np.linspace(-3, 3, 1000)
X, Y = np.meshgrid(x, y)
Z = f_multivariada([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.xlabel("$x$")
plt.ylabel("$y$")
plt.tick_params(labelsize = 8)
plt.show()
```



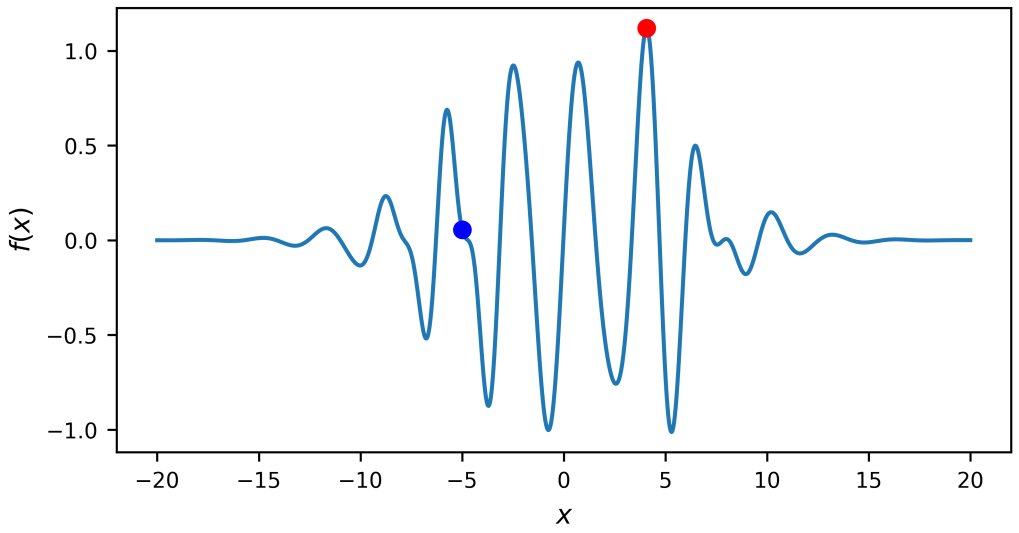
[7]:
```python
def simulated_annealing(f, x_actual, num_iter, tamano_paso = 1.0, t_actual =␣
↪100, alpha = 0.95):
    x_historial = [x_actual]
    f_actual = f(x_actual)
    x_mejor = x_actual
    f_mejor = f_actual
    for _ in range(num_iter):
        # Vecino aleatorio
        if isinstance(x_actual, np.ndarray):
            dx = np.random.normal(0, tamano_paso, x_actual.shape[0])
        else:
            dx = np.random.normal(0, tamano_paso)
        x_nuevo = x_actual + dx
        f_nuevo = f(x_nuevo)
        x_historial.append(x_nuevo)
        # Cambio en la función objetivo
        delta = f_nuevo - f_actual
        if delta > 0:
            x_actual = x_nuevo
            f_actual = f_nuevo
            # Actualizamos el mejor punto
            if f_actual > f_mejor:
                x_mejor = x_actual
                f_mejor = f_actual
        else:
            prob_aceptacion = np.exp(delta / t_actual)
            if np.random.rand() < prob_aceptacion:
                x_actual = x_nuevo
```

```
                f_actual = f_nuevo
            t_actual = alpha * t_actual
        x_historial = np.array(x_historial)
        return x_mejor, f_mejor, x_historial
```

[8]:
```
# Ejecucion
x_actual = -5
resultado_uni = simulated_annealing(f_univariada, x_actual, num_iter = 10000)
print("x =", resultado_uni[0])
print("f(x) =", resultado_uni[1])
```

```
x = 4.07667992396672
f(x) = 1.11871866141284
```

[9]:
```
# Graficar la función
x = np.linspace(-20, 20, 1000)
y = f_univariada(x)
plt.figure(figsize = (6, 3), dpi = resolucion)
plt.plot(x, y)
#plt.plot(resultado_uni[2][:, 0], f_univariada(resultado_uni[2][:, 0]), "oo")
plt.plot(resultado_uni[2][0], f_univariada(resultado_uni[2][0]), "bo")
plt.plot(resultado_uni[0], resultado_uni[1], "ro")
plt.xlabel("$x$")
plt.ylabel("$f(x)$")
plt.tick_params(labelsize = 8)
plt.show()
```

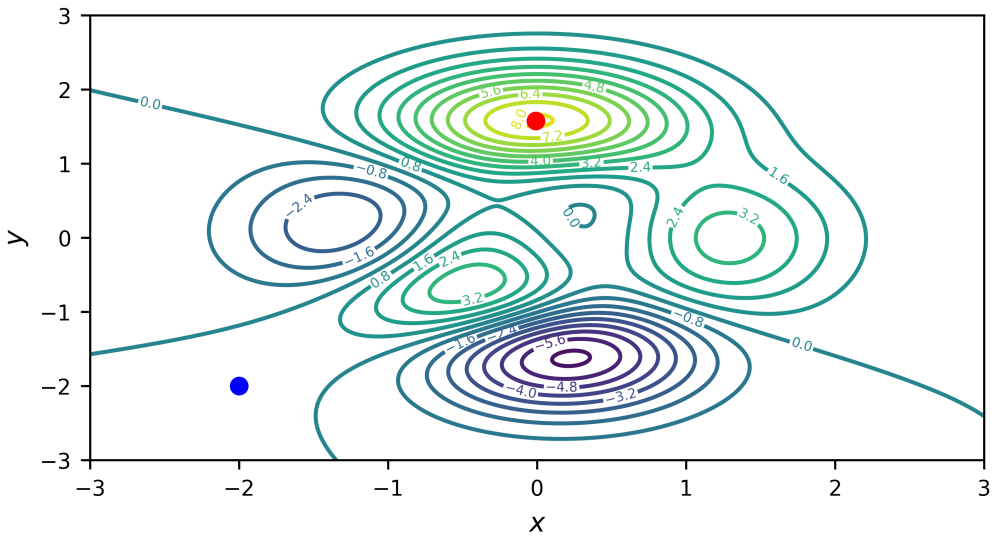


[10]:
```
# Ejecucion
x_actual = np.array([-2, -2])
```

```
resultado_mul = simulated_annealing(f_multivariada, x_actual, num_iter = 10000)
print("x =", resultado_mul[0])
print("f(x, y) =", resultado_mul[1])
```

```
x = [-0.00773645  1.57550947]
f(x, y) = 8.105631701252456
```

[11]:
```
x = np.linspace(-3, 3, 1000)
y = np.linspace(-3, 3, 1000)
X, Y = np.meshgrid(x, y)
Z = f_multivariada([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.plot(resultado_mul[2][:, 0], resultado_mul[2][:, 1], "ro")
plt.plot(resultado_mul[2][0, 0], resultado_mul[2][0, 1], "bo")
plt.plot(resultado_mul[0][0], resultado_mul[0][1], "ro")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.tick_params(labelsize = 8)
plt.show()
```



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

Universidad de Atacama, Facultad de Ingeniería, Departamento de Matemática