Universidad Nacional Autónoma de México

Facultad de Ciencias

Cómputo Evolutivo - 7140

T A R E A 2

Equipo:

-

Castillo Hernández Antonio - 320017438

☎☎

Luna Campos Emiliano - 320292084

☎☎

Juárez Cruz Joshua - 320124516

☎☎

-

Fecha de entrega:
17 de septiembre del 2025

Profesor:
Oscar Hernández Constantino

Ayudantes:
Rodrigo Fernando Velázquez Cruz
María Adelina Torres Ochoa

Tarea 2 - Búsqueda Local para Optimización Continua

1. **Funciones de Prueba**

   Considera los siguientes problemas de prueba de optimización continua.

   a) **Sphere**

$$f(x) = \sum_{i=1}^{n} x_i^2, \quad x_i \in [-5.12, +5.12]$$

   b) **Ackley**

$$f(x) = 20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right), \quad x_i \in [-30, +30]$$

   c) **Griewank**

$$f(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right), \quad x_i \in [-600, +600]$$

   d) **Rastrigin**

$$f(x) = 10n + \sum_{i=1}^{n}\left[x_i^2 - 10\cos(2\pi x_i)\right], \quad x_i \in [-5.12, +5.12]$$

   e) **Rosenbrock**

$$f(x) = \sum_{i=1}^{n-1}\left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right], \quad x_i \in [-2.048, +2.048]$$

   a) Da una breve descripción/discusión de sus características para cada una de las funciones.

   b) ¿Cuál podría ser más fácil y difícil de optimizar?, ¿Por qué?

   c) ¿La dimensión puede alterar (ya sea incrementar o disminuir) la dificultad de la optimización?

   a) **Funciones de Prueba**

      1) **Descripción de las funciones**

         - **Sphere:** Es una función convexa y suave. Presenta un único mínimo global en el origen y no tiene complicaciones adicionales.
         - **Ackley:** Presenta muchos mínimos locales, aunque todos dentro de una gran cuenca que conduce al mínimo global en el origen. Se utiliza para evaluar la capacidad de un algoritmo de escapar de trampas locales.
         - **Griewank:** Tiene un paisaje con valles y colinas distribuidos regularmente. El mínimo global está en el origen y su dificultad es moderada.
         - **Rastrigin:** Tiene un gran número de mínimos locales causados por la componente coseno. Aunque el mínimo global está en el origen, el entorno es complejo y puede confundir a los algoritmos de búsqueda.
         - **Rosenbrock:** Su mínimo global está en $(1, 1, \ldots, 1)$. La dificultad proviene de un valle largo y estrecho que hace complicada la convergencia hacia la solución.

2) **Más fácil y más difícil de optimizar**
La más fácil es **Sphere**, ya que es simple, convexa y no presenta obstáculos. Las más difíciles son **Rastrigin**, por la gran cantidad de mínimos locales, y **Rosenbrock**, debido a su valle curvado y estrecho. La **Ackley** también puede considerarse complicada, aunque menos que Rastrigin.

3) **Efecto de la dimensión**
La dificultad aumenta cuando crece la dimensión:
   - En funciones simples como **Sphere**, solo se incrementa la carga computacional.
   - En **Ackley**, **Griewank** y **Rastrigin**, el número de mínimos locales se multiplica y complica la búsqueda del óptimo global.
   - En **Rosenbrock**, el valle se vuelve más largo y difícil de recorrer.

Implementar las funciones anteriores: Cada función deberá estar implementada en un método (o función) diferente. En todos los casos, se deberá considerar que el parámetro de entrada será el valor del punto a evaluar x (arreglo de números reales). Dependiendo del lenguaje y la implementación, se puede recibir también la dimensión del problema.

Sphere:

```
def sphere(x : np.array):
  if np.any(x < -5.12) or np.any(x>5.12):
      raise ValueError("Advertencia: algunos valores están
      fuera del rango [-5.12, 5.12]")
  return np.sum(x**2.0)
```

Ackley

```
def ackley(X : np.array, a=20, b=0.2, c=2*np.pi):
  if np.any(X < -30) or np.any(X > 30):
      raise ValueError("Advertencia: algunos valores están fuera del rango [-30, 30]")

  n = len(X)

  sum_squares = np.sum(X**2.0)

  sum_cos = np.sum(np.cos(c * X))
  term_1 = -a*np.exp(-b * np.sqrt(sum_squares/n))
  term_2 = -np.exp(sum_cos/n)
  return a + np.e + term_1 + term_2
```

Griewank

```
  def griewank(X : np.array):
      if np.any(X < -600) or np.any(X > 600):
          raise ValueError("Advertencia: algunos valores
          están fuera del rango [-600, 600]")

      n = len(X)
```

```
    sum_term = np.sum(X**2.0) / 4000.0

    indices = np.arange(1, n+1)
    cos_terms = np.cos(X / np.sqrt(indices))
    prod_term = np.prod(cos_terms)

    return 1 + sum_term + prod_term
```

Rastrigin

```
def rastrigin(X : np.array):
 if np.any(X < -5.12) or np.any(X > 5.12):
     raise ValueError("Advertencia: algunos valores están
     fuera del rango [-5.12, 5.12]")

 n = len(X)
 sum_term = np.sum(X**2.0 - 10.0*np.cos(2.0 * np.pi * X))

 return 10*n + sum_term
```

Rosenbrock

```
 def rosenbrock(X : np.array):
 if np.any(X < -2.048) or np.any(X > 2.048):
     raise ValueError("Advertencia: algunos
     valores están fuera del rango [-2.048, 2.048]")
 sum_term = np.sum(100.0 * (X[1:] - X[:-1]**2.0)**2.0 + (1 - X[:-1])**2.0)
 return sum_term
```

Realmente sobre estas implementaciones no hay mucho que abordar. Simplemente se hizo uso de una paquetería para facilitar la representación de las distintas operaciones. Básicamente tomamos las funciones en papel y las pusimos en maquina.

2. **Representación Binaria**

Existen diferentes métodos para codificar números reales, por ejemplo: números de punto flotante, representación binaria, representación de punto fijo y código de Gray.

a) Describe e implementa el algoritmo de representación binaria para mapear (codificar y decodificar) números naturales. Menciona cuántos números son representables con **m** bits.

Un número natural puede representarse en binario usando m bits.

- El valor mínimo representable es 0.
- El valor máximo representable es $2^m - 1$.
- En total, con m bits se pueden representar 2^m números distintos.

Codificación: dividir el número repetidamente entre 2 y tomar los residuos.

Decodificación: multiplicar cada bit por su potencia de 2 correspondiente y sumar.

(Códigos en la carpeta src.)

b) Si se requiere una representación uniforme de números reales en el intervalo $[a, b]$: ¿Cómo se generaliza la representación binaria para mapear números reales?, ¿Cuál es la máxima precisión?

Para representar un número real en un intervalo $[a, b]$ usando m bits, se hace una partición uniforme en 2^m subintervalos.

El mapeo se define como:

$$x_{\text{real}} = a + \frac{k}{2^m - 1}(b - a), \quad k \in \{0, 1, \ldots, 2^m - 1\}.$$

La **precisión máxima** o tamaño de paso es:

$$\Delta = \frac{b - a}{2^m - 1}.$$

c) Implementa un algoritmo que codifique números reales en una representación binaria, considerando una partición uniforme sobre un intervalo $[a, b]$, utilizando m bits.
Toma el máximo valor que se puede obtener, Y se aplica una linealización sobre la que se guarda el numero real.

```
def codifica(x, n_bits, a, b):
     max_val = (1 << n_bits) - 1  # 2^n_bits - 1
     k = round((x - a) * max_val / (b - a))
     return codifica_dec(k)
```

d) Implementa un algoritmo para decodificar los vectores de bits como un número real.
Este mapeo funciona tomando una representación de bits y colocandola en un intervalo a-b. Primero recorre los bits y los transforma en un natural; posteriormente se realiza una transformación lineal, se busca el máximo valor con los bits dados y el intervalo se divide entre este valor máximo, para obtener un punto intermedio.

```
def decodifica(x_cod, n_bits, a, b):
    k = 0
    for i in range(n_bits):
        k = (k << 1) | x_cod[i]

    max_val = (1 << n_bits) - 1
    return a + k * (b - a) / max_val
```

e) Implementa las funciones necesarias para codificar y decodificar vectores de números reales.
No hay misterio en estas dos, simplemente recorremos los vectores y aplicamos las funciones creadas arriba

```
    def codifica_array(x, dim_x, n_bits, a, b):
    res = [0] * (dim_x * n_bits)
    for i in range(dim_x):
        bits = codifica(x[i], n_bits, a, b)
        for j in range(n_bits):
            res[i * n_bits + j] = bits[j]

    return res


def decodifica_array(x_cod, dim_x, n_bits, a, b):
    res = [0.0] * dim_x
    for i in range(dim_x):
        bits = [0] * n_bits
        for j in range(n_bits):
            bits[j] = x_cod[i * n_bits + j]
        res[i] = decodifica(bits, n_bits, a, b)
    return res
```

3. **Búsqueda Local**

   Considerando soluciones representadas como vectores (arreglos) de bits:

   - Describe e implementa un generador de soluciones aleatorias.
   - Describe e implementa una función u operador de vecindad.
   - Implementa el algoritmo de **Búsqueda por Descenso** con sus tres variantes:
     a) Mayor descenso
     b) Descenso aleatorio
     c) Primer descenso

   Experimentación: Ejecutar al menos 10 veces cada algoritmo (variante). Probar al menos con dimensión 10 (sugerencia: también con dimensión 2 y 5). Registrar resultados en una tabla como la siguiente:

   Realizamos una clase *Búsqueda Local*,la cual contiene las funciones anteriores.

   En esta se encuentra un constructor que recibe entre otros argumentos, la función que se busca minimizar, la dimensión del problema y los bits para la representación.

   Así pues tenemos métodos que resuelven la generación de soluciones aleatorias y la generación de una vecindad.

```
def generar_vecindad(self, solucion):
  vecinos = []
  filas, cols = solucion.shape

  for i in range(filas):
```

```
        for j in range(cols):
            vecino = solucion.copy()
            vecino[i, j] = 1 - vecino[i, j]  # Flip: 0->1, 1->0
            vecinos.append(vecino)
    return vecinos
```

Dado el vector (en este caso matriz) se iteran las posiciones cambiando por un 1 o un 0 segun sea el caso

El segundo resulta ser mas trivial de lo que parece, simplemente utilizamos un random del tamaño de la dimensión y numero de bits.

```
def generar_solucion_aleatoria(self):
return np.random.randint(0, 2, size=(self.dimension, self.bits_por_var))
```

Ahora bien, implementamos tres métodos; uno para cada variante de $BusquedaporDescenso$

Este es el desarrollado para mayor descenso, recorre toda la lista de vecinos y se queda con el mejor.

```
 for iteracion in range(max_iter):
    vecinos = self.generar_vecindad(solucion_actual)
    mejor_vecino = None
    mejor_fitness = fitness_actual

    for vecino in vecinos:
        fitness_vecino = self.evaluar_solucion(vecino)
        evaluaciones += 1

        if fitness_vecino < mejor_fitness:
            mejor_vecino = vecino
            mejor_fitness = fitness_vecino
```

Este es para Primer Descenso, que encuentra el primer mejor vecino y lo toma.

```
for iteracion in range(max_iter):
    vecinos = self.generar_vecindad(solucion_actual)

    mejor_vecino = None
    mejor_fitness = fitness_actual

    for vecino in vecinos:
        fitness_vecino = self.evaluar_solucion(vecino)
        evaluaciones += 1

        if fitness_vecino < fitness_actual:
            mejor_vecino = vecino
```

```
                mejor_fitness = fitness_vecino
                break
```

A primera vista prodian parecer el mismo, sin embargo la diferencia radica en el *break* dentro del if; primer vecino lo tiene, lo que hace que cuando encuentre un mejor vecino no vuelva a iterar, sino que se quede con el que tiene.

Finalmente, la versión aleatoria que escoge aleatoriamente al mejor vecino con random.shuffle

```
        for iteracion in range(max_iter):
            vecinos = self.generar_vecindad(solucion_actual)
            random.shuffle(vecinos)

            mejor_vecino = None
            mejor_fitness = fitness_actual

            for vecino in vecinos:
                fitness_vecino = self.evaluar_solucion(vecino)
                evaluaciones += 1

                if fitness_vecino < fitness_actual:
                    mejor_vecino = vecino
                    mejor_fitness = fitness_vecino
                    break
```

Dadas las 3 variantes las aplicamos a las cinco funciones del primer ejercicio un total de diez veces, acomodando en los parámetros una dimensión de 10, un numero de bits de 10, y sus respectivos limites de evaluación.

Colocamos asi para cada caso el mejor valor obtenido en las 10 iteraciones, el peor obtenido en las 10 iteraciones y el promedio a traves de la media aritmetica.

Esta es la tabla para la muestra de descenso aleatorio

Función	Dimensión	num_bits	Mejor valor $f(x)$	Valor promedio $f(x)$	Peor valor $f(x)$
Sphere	10	10	0.000250	0.4568617	2.533997
Ackley	10	10	237.506104	349.199606	521.926818
griewank	10	10	0.438087	7.289930	32.890474
rastrigin	10	10	17.401569	29.024964	40.515890
rosenbrock	10	10	56.554228	184.521160	458.012734

Esta es la tabla para la muestra de primer descenso

Función	Dimensión	num_bits	Mejor valor $f(x)$	Valor promedio $f(x)$	Peor valor $f(x)$
Sphere	10	10	0.000250	0.000250	0.000250
Ackley	10	10	246.360382	349.199606	521.926818
griewank	10	10	0.376013	34.860815	115.061194
rastrigin	10	10	11.488916	19.093293	30.272586
rosenbrock	10	10	117.381586	1802.648704	4361.730364

Esta es la tabla para la muestra de mayor descenso

Función	Dimensión	num_bits	Mejor valor $f(x)$	Valor promedio $f(x)$	Peor valor $f(x)$
Sphere	10	10	0.000250	0.0003903	0.001653
Ackley	10	10	246.360382	349.199606	521.926818
griewank	10	10	0.258885	0.523603	0.760685
rastrigin	10	10	14.842604	22.400393	32.979432
rosenbrock	10	10	7.611885	28.445476	96.274385

Referencias

- Surjanovic, S., & Bingham, D. (2013). *Sphere Function*. Recuperado el 15 de septiembre de 2025, de `https://www.sfu.ca/~ssurjano/spheref.html`
- Surjanovic, S., & Bingham, D. (2013). *Ackley Function*. Recuperado el 15 de septiembre de 2025, de `https://www.sfu.ca/~ssurjano/ackley.html`
- Surjanovic, S., & Bingham, D. (2013). *Griewank Function*. Recuperado el 15 de septiembre de 2025, de `https://www.sfu.ca/~ssurjano/griewank.html`
- Surjanovic, S., & Bingham, D. (2013). *Rastrigin Function*. Recuperado el 15 de septiembre de 2025, de `https://www.sfu.ca/~ssurjano/rastr.html`
- Surjanovic, S., & Bingham, D. (2013). *Rosenbrock Function*. Recuperado el 15 de septiembre de 2025, de `https://www.sfu.ca/~ssurjano/rosen.html`