2025/06/16

PROYECTO FINAL

- Marco Antonio Molina Mendoza
- Sabdi Rivera Tavera
- Jessica Beatriz Arcos Gutiérrez
- Angelica Velazquez Rodriguez

Contenido

1

Introducción

La optimización no lineal constituye una herramienta fundamental en matemáticas aplicadas, ingeniería e inteligencia artificial, donde la búsqueda de soluciones óptimas permite resolver problemas complejos en campos como economía, diseño de sistemas y aprendizaje automático. Sin embargo, la diversidad de problemas de optimización (desde funciones suaves hasta paisajes multimodales con múltiples óptimos locales) plantea un desafío crítico: *no existe un algoritmo universalmente superior*. La elección del método depende de factores como la naturaleza de la función objetivo, el costo computacional, la disponibilidad de derivadas y la precisión requerida.

Motivación

Este proyecto surge de la necesidad de evaluar científicamente el rendimiento de métodos de optimización en escenarios realistas. Muchos estudios comparativos se limitan a pruebas con puntos iniciales únicos, lo que puede sesgar los resultados. Aquí, implementamos una metodología rigurosa mediante **técnicas de multistart** con 2,500 puntos iniciales distribuidos uniformemente, garantizando robustez estadística y reproducibilidad. Además, seleccionamos siete funciones de prueba (seis multimodales y una cuadrática) que representan desafíos típicos en aplicaciones científicas y técnicas, desde la clásica Booth hasta la compleja Goldstein-Price.

Contribución

Este trabajo no solo valida teóricamente los métodos estudiados, sino que ofrece un *análisis práctico* mediante implementaciones en MATLAB, código reproducible y resultados detallados para cada función. Los hallazgos destacan *trade-offs* entre velocidad, precisión y requerimientos computacionales, demostrando, por ejemplo, la eficiencia de Newton en funciones cuadráticas frente a la resiliencia de Hooke-Jeeves en problemas sin derivadas. Esta comparación sistemática sirve como referencia para investigadores e ingenieros que enfrentan problemas de optimización en entornos multidisciplinarios.

2

Marco teórico

2.1 Método de descenso con máxima pendiente

El método de descenso con máxima pendiente, también conocido como método de Cauchy o steepest descent, es un algoritmo iterativo ampliamente utilizado en optimización no lineal. Su objetivo es encontrar el mínimo local de una función diferenciable en $\mathbb{R}^n$.

En cada iteración, el método de descenso requiere calcular una dirección de búsqueda v y un tamaño de paso α, actualizando el punto actual $x^{(k)}$ a un nuevo punto $x^{(k+1)}$ mediante la fórmula

$$x^{(k+1)} = x^{(k)} + \alpha v.$$

La dirección de descenso se elige como el negativo del gradiente, $-\nabla f\left(x^{(k)}\right)$, ya que esta es la dirección de máximo decremento de la función. El tamaño de paso α_k se determina resolviendo un problema de minimización unidimensional a lo largo de la dirección de descenso, es decir,

$$\alpha_k = \arg\min_{\alpha \geq 0} \left\{ f\left(x^{(k)} + \alpha_k v_k\right) \right\}.$$

Este proceso se repite hasta que se alcanza un punto óptimo local.

2.1.1 Condiciones de Wolfe

Las Condiciones de Wolfe son un conjunto de criterios para asegurar que el tamaño de paso en cada iteración conduzca a una reducción suficiente de la función objetivo y a un progreso adecuado hacia el mínimo. Las Condiciones de Wolfe constan de dos partes:

- Condición de Armijo: Esta condición dada por $f\left(x^{(k)} + \alpha_k v_k\right) \leq f\left(x^{(k)}\right) + c_1 \alpha_k \nabla f\left(x^{(k)}\right)^\top v_k$, donde $0 \leq c_1 \leq 1$, asegura que la disminución en la función objetivo sea proporcional al tamaño del paso y la derivada direccional.
- Condición de curvatura: Esta condición dada por $\nabla f\left(x^{(k)} + \alpha_k v_k\right)^\top v_k \geq c_2 \nabla f\left(x^{(k)}\right)^\top v_k$, donde $c_1 \leq c_2 \leq 1$, asegura que el tamaño del paso no sea demasiado pequeño.

2.2 Método de Newton para varias variables

El método de Newton se basa en aproximar la función objetivo mediante una expansión cuadrática de Taylor alrededor del punto actual $x^{(k)}$. Esta aproximación tiene en cuenta tanto el gradiente $\nabla f\left(x^{(k)}\right)$ como la matriz Hessiana $H\left(x^{(k)}\right)$. El objetivo es minimizar la siguiente función,

$$q(x) = f\left(x^{(k)}\right) + \left(x - x^{(k)}\right)^{\top} g^{(k)} + \frac{1}{2}\left(x - x^{(k)}\right)^{\top} H\left(x^{(k)}\right)\left(x - x^{(k)}\right)$$

donde $g^{(k)} = \nabla f\left(x^{(k)}\right)$ y $H := \nabla^2 f\left(x^{(k)}\right)$, que es una aproximación cuadrática a la función original. El mínimo de dicha función se encuentra cuando

$$H\left(x^{(k)}\right)^{-1} g^{(k)} + \left(x - x^{(k)}\right) = 0.$$

De esta manera, la nueva iteración se calcula como

$$x^{(k+1)} = x^{(k)} - H\left(x^{(k)}\right)^{-1} g^{(k)}.$$

Si la matriz Hessiana es definida positiva en el punto $x^{(k)}$, entonces $H\left(x^{(k)}\right)^{-1} g^{(k)}$ nos llevará a un mínimo local de la función. El método de Newton no siempre garantiza el descenso; es decir, no siempre se cumple que $f\left(x^{(k+1)}\right) < f\left(x^{(k)}\right)$. Además, calcular la Hessiana y resolver el sistema lineal en cada iteración puede ser computacionalmente costoso, especialmente para problemas de gran escala. La Hessiana también puede ser singular, lo que impide la inversión.

2.2.1 Modificación Levenberg-Marquardt

Para mitigar algunos de los problemas del método de Newton, como la falta de definitud positiva de la matriz Hessiana, se introduce una modificación llamada modificación de Levenberg-Marquardt. Esta técnica ajusta la matriz Hessiana de la siguiente manera

$$x^{(k+1)} = x^{(k)} - \left[H\left(x^{(k)}\right) + \mu_k I\right]^{-1} g^{(k)},$$

donde μ_k es un parámetro que se ajusta durante el proceso. Si μ_k es grande, el comportamiento del método se asemeja al del método de máxima pendiente con un paso pequeño, mientras que si μ_k es pequeño, el algoritmo se aproxima al método de Newton.

2.3 Método de Cuasi-Newton

El Método Cuasi-Newton es una modificación del método de Newton que busca aproximar la inversa de la matriz Hessiana de la función objetivo de forma iterativa. Esto se hace para reducir el costo computacional de calcular la Hessiana en cada iteración, como lo requiere el método de Newton, y para evitar problemas cuando la Hessiana es singular o no definida positiva. En cada iteración, se añade una matriz de corrección U_k a la matriz H_k para obtener la siguiente aproximación:

$$H_{k+1} = H_k + U_k.$$

Esta actualización se realiza para incorporar información sobre la curvatura de la función objetivo, basándose en los cambios observados en los gradientes. La condición de Cuasi-Newton, derivada de la expansión de Taylor de segundo orden, establece que

$$g_{k+1} - g_k \approx H(x_k)d_k$$

donde $g_{k+1} := \nabla f(x_k + d_k)$ y $d_k = x_{k+1} - x_k$ es el paso dado en la iteración.

2.3.1 Rango uno

El método de rango uno es un método Cuasi-Newton específico que calcula la actualización de la inversa de la Hessiana mediante la siguiente fórmula:

$$H_{k+1} = H_k + \frac{\left(\Delta x^{(k)} - H_k \Delta g^{(k)}\right)\left(\Delta x^{(k)} - H_k \Delta g^{(k)}\right)^\top}{\Delta\, g^{(k)\,\top}\left(\Delta x^{(k)} - H_k \Delta g^{(k)}\right)}$$

para obtener $d^{(k)} = -H_k g^{(k)}$ donde

$$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$$

y α_k es el tamaño de paso.

2.3.2 Rango dos

El método de rango dos (DFP) es un método Cuasi-Newton que actualiza la aproximación de la inversa de la Hessiana garantizando que si H_k es definida positiva, entonces H_{k+1} también lo será. La fórmula de actualización es:

$$H_{k+1} = H_k + \frac{\Delta x^{(k)} \Delta x^{(k)\top}}{\Delta x^{(k)\top} \Delta g^{(k)}} - \frac{H_k \Delta g^{(k)} \Delta g^{(k)\top} H_k}{\Delta g^{(k)\top} H_k \Delta g^{(k)}}$$

donde: $\Delta x^{(k)} = x^{(k+1)} - x^{(k)}$ (cambio en el punto), $\Delta g^{(k)} = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)})$ (cambio en el gradiente), H_k es la aproximación actual de la inversa de la Hessiana.

2.4 Método de gradientes conjugados

El método de Gradientes Conjugados es un algoritmo iterativo para resolver sistemas de ecuaciones lineales y optimización no lineal. Es particularmente eficiente para minimizar funciones cuadráticas y puede extenderse a funciones no cuadráticas.

2.4.1 Fórmulas para β_k

- Hestenes-Stiefel: $\beta_k^{HS} = \dfrac{\nabla f(x^{k+1})^T(\nabla f(x^{k+1}) - \nabla f(x^k))}{d^{kT}(\nabla f(x^{k+1}) - \nabla f(x^k))}$.
- Polak-Ribière: $\beta_k^{PR} = \dfrac{\nabla f(x^{k+1})^T(\nabla f(x^{k+1}) - \nabla f(x^k))}{\nabla f(x^k)^T \nabla f(x^k)}$.
- Fletcher-Reeves: $\beta_k^{FR} = \dfrac{\nabla f(x^{k+1})^T \nabla f(x^{k+1})}{\nabla f(x^k)^T \nabla f(x^k)}$.

3 Metodología

La metodología de este proyecto se centra en la evaluación de diferentes métodos de optimización no lineal sin restricciones. Para ello, se busca el punto óptimo de diversas funciones objetivo utilizando cinco métodos distintos: Descenso con Máxima Pendiente, Método de Newton, Método Cuasi-Newton, Gradientes Conjugados y el algoritmo de Hooke-Jeeves.

Para garantizar la robustez y fiabilidad de los resultados, se implementará la técnica de *multistart*. En lugar de evaluar cada método desde un único punto inicial, se generan N[1] puntos iniciales aleatorios uniformemente distribuidos dentro del dominio definido para cada función. Esto permite mitigar el riesgo de obtener resultados sesgados por la elección de un único punto de partida, que podría tener ventajas específicas o conducir a óptimos locales. Para asegurar la reproducibilidad de los experimentos, se controla el generador de números aleatorios estableciendo una semilla fija en el código, de modo que cada conjunto de puntos iniciales se utilice de manera consistente para cada método y función.

Una vez generados los N puntos iniciales, cada uno de los cinco métodos de optimización se ejecuta para cada punto. De cada ejecución, se recopilan los siguientes resultados:

- Número de iteraciones: Este indicador refleja el costo computacional del método en términos de evaluaciones de la función objetivo.
- Punto óptimo encontrado: Este valor es crucial para verificar la correcta convergencia del método y para identificar si algún punto inicial no converge a la solución esperada.
- Función evaluada en el punto óptimo: Este valor es el resultado de sustituir el punto óptimo encontrado en la función objetivo, lo que indica el valor mínimo (o máximo) alcanzado por la función.
- Error: Este parámetro mide la norma de la diferencia entre el punto óptimo encontrado por el método y el verdadero punto óptimo de la función, lo que permite evaluar la precisión del algoritmo.

Finalmente, se realizará un análisis comparativo de los resultados obtenidos para cada método. Este análisis permitirá evaluar la efectividad, eficiencia y fiabilidad de cada algoritmo en la resolución de problemas de optimización.

[1] Como en nuestras anteriores tareas, asignaremos 2500 al valor de N.

3.1 Programas de MATLAB

3.1.1 Algoritmo de Máxima Pendiente

```
1   unction [best_x, best_val, tiempo_ejecucion, ...
        num_evaluaciones, x_inicios, resultados] = ...
        multistart_maxima_pen_interv(f, grad_f, num_inicios, lb, ub)
2   % Multistart con descenso por máxima pendiente y registro de ...
        puntos iniciales
3   % Con error y número de iteraciones
4   % lb: límite inferior
5   % ub: límite superior
6
7   tic;
8   num_evaluaciones = 0;
9   best_val = Inf;
10  best_x = [];
11  x_inicios = zeros(2, num_inicios);
12  resultados = struct('x0', {}, 'x_opt', {}, 'f_val', {}, ...
        'evals', {}, 'iterations', {}, 'error', {});
13
14  rng(42); % Fija semilla aleatoria para reproducibilidad
15
16  for i = 1:num_inicios
17
18      x0 = lb + (ub - lb) .* rand(2,1);
19      x_inicios(:, i) = x0;
20
21      % Optimización desde x0
22      [x_opt, evals, iterations, err] = maxima_pen_interv(f, ...
            grad_f, x0);
23      val = f(x_opt);
24      num_evaluaciones = num_evaluaciones + evals + 1; % +1 por ...
            evaluación final
25
26      % Guarda resultados individuales
27      resultados(i).x0 = x0;
28      resultados(i).x_opt = x_opt;
```

```
29          resultados(i).f_val = val;
30          resultados(i).evals = evals;
31          resultados(i).iterations = iterations;
32          resultados(i).error = err;
33
34          % Actualiza mejor solución global
35          if val < best_val
36              best_val = val;
37              best_x = x_opt;
38          end
39      end
40      tiempo_ejecucion = toc;
41  end
42
43  % ...
        ----------------------------------------------------------------------
44  function [x_opt, num_evaluaciones, iterations, error] = ...
        maxima_pen_interv(f, grad_f, x0)
45      % Descenso por gradiente con búsqueda lineal exacta
46      % Número de iteraciones y error final
47
48      tol = 1e-6;
49      max_iter = 1000;
50      iterations = 0;
51      xk = x0(:);
52      num_evaluaciones = 0;
53      error = Inf;
54
55      while norm(grad_f(xk)) > tol && iterations < max_iter
56          vk = -grad_f(xk); % Dirección de descenso
57          % Búsqueda lineal (fminbnd usa ~30 evaluaciones internas)
58          alpha_k = fminbnd(@(alpha) f(xk + alpha * vk), 0, 1);
59          xk = xk + alpha_k * vk;
60          iterations = iterations + 1;
61          num_evaluaciones = num_evaluaciones + 30; % Estimación ...
                conservadora
62          error = norm(grad_f(xk));
63      end
```

```
64      x_opt = xk;
65  end
66
67  % ...
        ------------------------------------------------------------------------
68  % Función
69  f1 = @(x) x(1)^2 + 2*x(2)^2 - 0.3*cos(3*pi*x(1)) - ...
        0.4*cos(4*pi*x(2)) + 0.7;
70  % Gradiente numérico
71  grad_f1 = @(x) [
72      (f1(x + [1e-6; 0]) - f1(x - [1e-6; 0])) / (2e-6);
73      (f1(x + [0; 1e-6]) - f1(x - [0; 1e-6])) / (2e-6)
74  ];
75
76  % ...
        ------------------------------------------------------------------------
77
78  num_reinicios = 30;
79  lb = [-100; -100]; % Límites inferiores
80  ub = [100; 100];   % Límites superiores
81  [best_x, best_val, tiempo, evals, x_inicios, resultados] = ...
        multistart_maxima_pen_interv(f1, grad_f1, num_reinicios, lb, ub);
82
83  % Mostrar resultados en consola
84  disp('=== RESULTADOS ===');
85  disp(['Mejor mínimo encontrado: [', num2str(best_x'), ']']);
86  disp(['Valor de la función: ', num2str(best_val)]);
87  disp(['Tiempo total: ', num2str(tiempo), ' segundos']);
88  disp(['Evaluaciones de f(x): ', num2str(evals)]);
89
90  % Mostrar estadísticas de iteraciones y errores
91  iterations_array = [resultados.iterations];
92  error_array = [resultados.error];
93  disp(['Iteraciones promedio por ejecución: ', ...
        num2str(mean(iterations_array))]);
94  disp(['Error promedio final: ', num2str(mean(error_array))]);
95  disp(['Máximo error final: ', num2str(max(error_array))]);
```

3.1.2 Algoritmo del Método de Newton

```
function newton_multivariable_multistart_2500_symbolic()
% Limpiar entorno
clear; clc;

% Definir variables simbólicas
syms x1 x2
vars = [x1; x2];

% Definir la función simbólica (Goldstein-Price simplificada)
f_expr = (1.5 - x1 + x1*x2)^2 + (2.25 - x1 + x1*x2^2)^2 + (2.625 ...
    - x1 + x1*x2^3)^2;
% Calcular gradiddente y Hessiana simbólicas
grad_f_sym = gradient(f_expr, vars);
H_f_sym = hessian(f_expr, vars);

% Convertir a funciones numéricas
f = matlabFunction(f_expr, 'Vars', {vars});
grad_f = matlabFunction(grad_f_sym, 'Vars', {vars});
H_f = matlabFunction(H_f_sym, 'Vars', {vars});

% Parámetros
max_iter = 100;
tol = 1e-6;
num_starts = 2500;
rango = [-5, 5];

% Inicialización del mejor resultado
mejor_valor = inf;
mejor_resultado = struct();

for intento = 1:num_starts
    x0 = rango(1) + (rango(2) - rango(1)) * rand(2,1);
    x = x0;
    x_prev = inf(size(x0));
    iter = 0;
    converged = false;
```

```
36          llamadas_funcion = 0;
37
38          while iter < max_iter && ~converged
39              g = grad_f(x); g = g(:);  % vector columna
40              H_current = H_f(x);
41
42              if iter > 0
43                  error = norm(x - x_prev);
44              else
45                  error = inf;
46              end
47
48              if iter > 0 && error < tol
49                  converged = true;
50                  break;
51              end
52
53              % Comprobar condición numérica Hessiana
54              if rcond(H_current) < 1e-12
55                  break;
56              end
57
58              x_prev = x;
59              d = -H_current \ g;
60              x = x + d;
61
62              iter = iter + 1;
63              llamadas_funcion = llamadas_funcion + 1;
64          end
65
66          valor_final = f(x);
67          llamadas_funcion = llamadas_funcion + 1;
68          grad_final = grad_f(x); grad_final = grad_final(:);
69
70          if valor_final < mejor_valor
71              mejor_valor = valor_final;
72              mejor_resultado.x0 = x0;
73              mejor_resultado.x_opt = x;
```

```
74                mejor_resultado.valor_funcion = valor_final;
75                mejor_resultado.gradiente = grad_final;
76                mejor_resultado.error = norm(x - x_prev);
77                mejor_resultado.iteraciones = iter;
78                mejor_resultado.llamadas_funcion = llamadas_funcion;
79            end
80        end
81
82        % Mostrar solo el mejor resultado
83        fprintf('Mejor resultado encontrado (entre %d puntos):\n', ...
              num_starts);
84        fprintf('Punto inicial:      [%.6f, %.6f]\n', ...
              mejor_resultado.x0(1), mejor_resultado.x0(2));
85        fprintf('Punto óptimo:       [%.6f, %.6f]\n', ...
              mejor_resultado.x_opt(1), mejor_resultado.x_opt(2));
86        fprintf('Valor función:      %.6f\n', ...
              mejor_resultado.valor_funcion);
87        fprintf('Gradiente óptimo:   [%.6f, %.6f]\n', ...
              mejor_resultado.gradiente(1), mejor_resultado.gradiente(2));
88        fprintf('Error final:        %.6f\n', mejor_resultado.error);
89        fprintf('Iteraciones:        %d\n', mejor_resultado.iteraciones);
90        fprintf('Llamadas función:   %d\n', ...
              mejor_resultado.llamadas_funcion);
91    end
```

3.1.3 Algoritmo de Cuasi-Newton Rango 1

```
1   % Parámetros comunes
2   a = -10; b = 10;              % Límites del dominio
3   n_points = 2500;              % Número de puntos iniciales
4   dim = 2;                      % Dimensión del problema
5   true_opt = [1, 3];            % Óptimo verdadero
6   max_iter = 1000;              % Máximo de iteraciones por ejecución
7   tol = 1e-6;                   % Tolerancia de convergencia
8
9   % Generar 2500 puntos iniciales uniformemente distribuidos
10  rng(1); % Fijar semilla para reproducibilidad
11  initial_points = a + (b-a)*rand(n_points, dim);
12
```

```
% Definir la función objetivo
objective_func = @(x) (x(1) + 2*x(2) - 7)^2 + (2*x(1) + x(2) - ...
    5)^2; % Función de la sección 4.2

% Llamar al optimizador
[opt_x, fval, best_iter, err] = quasi_newton_rango1_ms(...
    objective_func, initial_points, a, b, true_opt, max_iter, tol);

% Mostrar resultados
fprintf('Cuasi-Newton Rango 1:\n');
fprintf('  Punto óptimo: [%.6f, %.6f]\n', opt_x(1), opt_x(2));
fprintf('  Valor función: %.6f\n', fval);
fprintf('  Iteraciones: %d\n', best_iter);
fprintf('  Error: %.6f\n', err);

% FUNCIÓN PRINCIPAL: Cuasi-Newton Rango 1 con MultiStart
function [best_x, best_fval, best_iter, best_err] = ...
    quasi_newton_rango1_ms(...
    f, initial_points, a, b, true_opt, max_iter, tol)

    n_points = size(initial_points, 1);
    dim = size(initial_points, 2);

    best_fval = inf;
    best_x = [];
    best_iter = 0;  % Almacena iteraciones de la mejor ejecución

    for i = 1:n_points
        x0 = initial_points(i,:);
        [x_opt, fval, iter, success] = quasi_newton_rango1(...
            f, x0, dim, a, b, max_iter, tol);

        % Guardar iteraciones SOLO si es la mejor ejecución
        if success && fval < best_fval
            best_fval = fval;
            best_x = x_opt;
            best_iter = iter;  % Guarda iteraciones de esta ejecución
        end
```

```
    end

    best_err = norm(best_x - true_opt);
end

% ALGORITMO DE CUASI-NEWTON RANGO 1
function [x_opt, fval, iter, success] = quasi_newton_rango1(...
    f, x0, dim, a, b, max_iter, tol)

    % Inicialización
    x = x0(:); % Convertir a vector columna
    H = eye(dim); % Matriz inicial identidad
    iter = 0;
    success = false;
    n_restart = 0; % Contador para reinicio

    % Almacenar puntos para evitar ciclos infinitos
    x_history = zeros(dim, max_iter+1);
    x_history(:,1) = x;

    while iter < max_iter
        iter = iter + 1;

        % Calcular gradiente con diferencias finitas
        g = gradiente(f, x);

        % Verificar convergencia
        if norm(g) < tol
            success = true;
            break;
        end

        % Reiniciar cada n iteraciones (n = dimensión)
        if n_restart >= dim
            d = -g; % Dirección de descenso más pronunciado
            H = eye(dim); % Reiniciar matriz H
            n_restart = 0; % Reiniciar contador
        else
```

```
87              % Calcular dirección de búsqueda
88              d = -H * g;
89              n_restart = n_restart + 1;
90          end
91
92          % Verificar dirección de descenso
93          if g' * d >= 0
94              d = -g; % Usar descenso más pronunciado
95              H = eye(dim); % Reiniciar matriz H
96              n_restart = 0;
97          end
98
99          % Búsqueda lineal exacta (sección dorada)
100         alpha = golden_section_search(f, x, d, a, b, 100);
101
102         % Actualizar punto
103         x_new = x + alpha * d;
104
105         % Asegurar que está dentro de los límites
106         x_new = max(min(x_new, b), a);
107
108         % Calcular nuevo gradiente
109         g_new = gradiente(f, x_new);
110
111         % Calcular diferencias
112         delta_x = x_new - x;
113         delta_g = g_new - g;
114
115         % Verificar condición de actualización
116         denom = delta_g' * (delta_x - H * delta_g);
117
118         % Actualizar matriz H si se cumple la condición
119         if abs(denom) > 1e-10
120             u = delta_x - H * delta_g;
121             H = H + (u * u') / denom;
122         end
123
124         % Verificar si la matriz H es definida positiva
```

```
125         if ~isdefinite(H)
126             % Si no es DP, usar descenso más pronunciado en la ...
                    próxima iteración
127             n_restart = dim; % Forzar reinicio
128         end
129 
130         % Actualizar para siguiente iteración
131         x = x_new;
132         x_history(:,iter+1) = x;
133 
134         % Verificar convergencia en posición
135         if norm(delta_x) < tol
136             success = true;
137             break;
138         end
139 
140         % Detectar estancamiento
141         if iter > 10 && norm(x - x_history(:,iter-5)) < tol
142             success = true;
143             break;
144         end
145     end
146 
147     x_opt = x';
148     fval = f(x_opt);
149 end
150 
151 
152 % FUNCIÓN PARA CALCULAR EL GRADIENTE (DIFERENCIAS FINITAS)
153 function g = gradiente(f, x)
154     h = 1e-8; % Tamaño del paso
155     n = length(x);
156     g = zeros(n, 1);
157 
158     for i = 1:n
159         e_i = zeros(n, 1);
160         e_i(i) = 1;
161 
```

```
162             f_forward = f(x + h*e_i);
163             f_backward = f(x - h*e_i);
164 
165             g(i) = (f_forward - f_backward) / (2*h);
166         end
167     end
168 
169     % BÚSQUEDA LINEAL CON MÉTODO DE LA SECCIÓN DORADA
170     function alpha = golden_section_search(f, x, d, a, b, max_iter)
171         % Parámetros de la sección dorada
172         golden_ratio = (sqrt(5)-1)/2;
173         tol = 1e-6;
174 
175         % Establecer intervalo inicial
176         low = 0;
177         high = 1;
178 
179         % Expandir intervalo si es necesario
180         while f(x + high*d) < f(x + low*d)
181             high = 2*high;
182         end
183 
184         % Puntos iniciales
185         x1 = high - golden_ratio*(high - low);
186         x2 = low + golden_ratio*(high - low);
187 
188         f1 = f(x + x1*d);
189         f2 = f(x + x2*d);
190 
191         % Iteraciones de la sección dorada
192         for i = 1:max_iter
193             if (high - low) < tol
194                 break;
195             end
196 
197             if f1 < f2
198                 high = x2;
199                 x2 = x1;
```

```
            f2 = f1;
            x1 = high - golden_ratio*(high - low);
            f1 = f(x + x1*d);
        else
            low = x1;
            x1 = x2;
            f1 = f2;
            x2 = low + golden_ratio*(high - low);
            f2 = f(x + x2*d);
        end
    end

    alpha = (low + high)/2;
end

% VERIFICAR SI UNA MATRIZ ES DEFINIDA POSITIVA
function result = isdefinite(A)
    % Se intenta realizar la descomposición de Cholesky
    [~, p] = chol(A);
    result = (p == 0);
end
```

3.1.4 Algoritmo del Método de Gradientes Conjugados

```
clc; clear;

fprintf('=== MultiInicio manual con gradientes conjugados ===\n');

% Variables simbólicas para la función y el gradiente
syms x y

%===============FUNCIONES A EVALUAR=================
% 1) 17. Función Bohachevsky: x(1)^2 + 2*x(2)^2 - ...
    0.3*cos(3*pi*x(1)) - 0.4*cos(4*pi*x(2)) + 0.7;
% 2) 24. Función Booth: (x(1) + 2*x(2) - 7)^2 + (2*x(1) + x(2) - ...
    5)^2;
% 3) 40. Goldstein-Price: (1 + (x(1) + x(2) + 1)^2 * (19 - ...
    14*x(1) + 3*x(1)^2 - 14*x(2) + 6*x(1)*x(2) + 3*x(2)^2)) * ...
%     (30 + (2*x(1) - 3*x(2))^2 * (18 - 32*x(1) + 12*x(1)^2 + ...
```

```
    48*x(2) - 36*x(1)*x(2) + 27*x(2)^2));
% 4) 25. Función Matyas: 0.26*(x(1)^2 + x(2)^2) - 0.48*x(1)*x(2);
% 5) 29. Función Three Hump Camel: 2*x(1)^2 - 1.05*x(1)^4 + ...
    (x(1)^6)/6 + x(1)*x(2) + x(2)^2;
% 6) 34. Función Eason: -cos(x(1))*cos(x(2))*exp(-(x(1)-pi)^2 - ...
    (x(2)-pi)^2);
% 7) 36. Función Beale:  @(x) (1.5 - x(1) + x(1)*x(2))^2 + (2.25 ...
    - x(1) + x(1)*x(2)^2)^2 + (2.625 - x(1) + x(1)*x(2)^3)^2;


funcion = @(x) x(1)^2 + 2*x(2)^2 - 0.3*cos(3*pi*x(1)) - ...
    0.4*cos(4*pi*x(2)) + 0.7;
gradiente = @(x) [
    2*x(1) + 0.9*pi*sin(3*pi*x(1));
    4*x(2) + 1.6*pi*sin(4*pi*x(2))
];

% Parámetros de MultiInicio
num_intentos = 2500;
lim_inf = [-100; -100];
lim_sup = [100; 100];

% Inicializar valores
% Inicializar valores
mejor_error = inf;  % Ahora el criterio será el error

for i = 1:num_intentos
    x_inicial = lim_inf + (lim_sup - lim_inf) .* rand(2,1);

    % Llamar al método del gradiente conjugado
    [x_optimo, gradiente_optimo, valor_funcion, iteraciones, ...
        llamadas_f] = gradiente_conjugado(x_inicial, funcion, ...
        gradiente);

    % Calcular el error actual
    error_actual = norm(x_optimo - x_inicial);

    % Comparar con el mejor error hasta ahora
```

```
44      if error_actual < mejor_error
45          mejor_error = error_actual;
46          mejor_valor = valor_funcion;
47          mejor_x0 = x_inicial;
48          mejor_xopt = x_optimo;
49          mejor_grad = gradiente_optimo;
50          mejor_iters = iteraciones;
51          mejor_llamadas = llamadas_f;
52      end
53  end
54
55
56
57  % Mostrar solo la mejor ejecución
58  fprintf('\n=== Mejor ejecución de %d intentos ===\n', num_intentos);
59  fprintf('Punto inicial: (%.6f, %.6f)\n', mejor_x0(1), mejor_x0(2));
60  fprintf('Punto óptimo:  (%.6f, %.6f)\n', mejor_xopt(1), ...
        mejor_xopt(2));
61  fprintf('Valor de la función: %.10f\n', mejor_valor);
62  fprintf('Gradiente en óptimo: (%.10f, %.10f)\n', mejor_grad(1), ...
        mejor_grad(2));
63  fprintf('Error ||x_opt - x0||: %.10f\n', mejor_error);
64  fprintf('Iteraciones:   %d\n', mejor_iters);
65  fprintf('Llamadas a f:  %d\n', mejor_llamadas);
66
67  %% Función: gradiente conjugado
68  function [x_optimo, gradiente_optimo, valor_funcion, iteraciones, ...
        llamadas_f] = gradiente_conjugado(x0, funcion, gradiente)
69      x_k = x0(:);
70      g_k = gradiente(x_k);
71      d_k = -g_k;
72      tolerancia = 1e-9;
73      iter_max = 3000;
74      iteraciones = 0;
75      llamadas_f = 0;
76
77      while iteraciones < iter_max
78          phi = @(alfa) funcion(x_k + alfa*d_k);
```

```
79          alfa_k = fminbnd(phi, 0, 1);
80          llamadas_f = llamadas_f + 1;
81
82          x_k1 = x_k + alfa_k*d_k;
83          g_k1 = gradiente(x_k1);
84          llamadas_f = llamadas_f + 1;
85
86          if norm(g_k1) < tolerancia
87              break;
88          end
89
90          if mod(iteraciones, length(x_k)) == 0
91              d_k1 = -g_k1;
92          else
93              beta_k = (g_k1'*g_k1) / (g_k'*g_k);
94              d_k1 = -g_k1 + beta_k * d_k;
95          end
96
97          x_k = x_k1;
98          g_k = g_k1;
99          d_k = d_k1;
100         iteraciones = iteraciones + 1;
101     end
102
103     x_optimo = x_k1;
104     gradiente_optimo = g_k1;
105     valor_funcion = funcion(x_optimo);
106     llamadas_f = llamadas_f + 1;
107 end
```

4

Funciones a optimizar

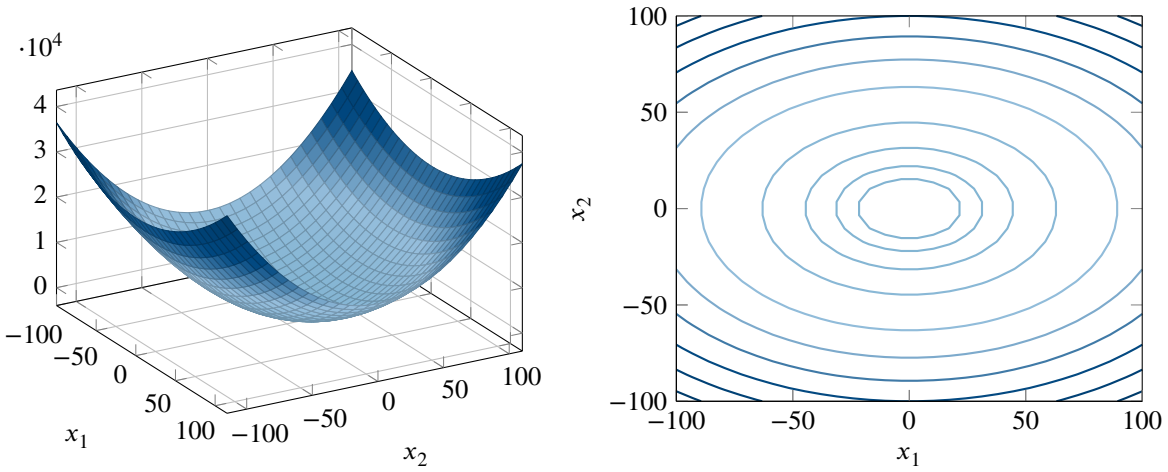
En este capítulo se presenta el núcleo experimental del proyecto: el análisis comparativo de cinco métodos de optimización aplicados a siete funciones de prueba. Se ha elegido un conjunto variado de funciones de prueba, considerando aspectos como la complejidad de su superficie, la presencia de múltiples óptimos locales, la dimensión del problema y la sensibilidad a las condiciones iniciales. Cada función ha sido seleccionada con el objetivo de cubrir un amplio espectro de escenarios que suelen encontrarse en problemas de optimización del mundo real.

4.1 Función de Bohachevsky

La función de Bohachevsky está definida como:

$$f_1(\mathbf{x}) = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7.$$

Está función la evaluaremos en $x_i \in [-100, 100]$, para $i = 1, 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (0, 0)$ y al evaluarlo en la función se obtiene $f_1(\mathbf{x}^*) = 0$.



(a) Gráfica de la función de Bohachevsky

(b) Curvas de nivel de la función de Bohachevsky

Figura 4.1

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	82	$(-4.5596 \times 10^{-9}, 4.1631 \times 10^{-10})$	0.000000	0.18034
Método de Newton	34	(0.00, 0.00)	0.00	0.00
Cuasi-Newton Rango 1	13	(−0.0000, −0.0000)	0.0000	0.0000
Gradientes Conjugados	279	(0.0000, 0.469528)	0.469882	0.039030
Hooke-Jeeves	65	(−0.000000, −0.000000)	0.000000	0.000000

Tabla 4.1 Resultados de la función de Bohachevsky usando *multistart* con $N = 2500$

El método **Cuasi-Newton de Rango 1** demuestra ser el más eficiente. Con tan solo **13 iteraciones**, converge al óptimo teórico (0, 0), obteniendo un valor de función y un error residual nulos.

Los métodos de **Newton** y **Hooke-Jeeves** también alcanzan el óptimo exacto con un error de cero. Sin embargo, requieren un esfuerzo computacional considerablemente mayor, con 34 y 65 iteraciones respectivamente, lo que los hace menos eficientes en comparación con el método Cuasi-Newton.

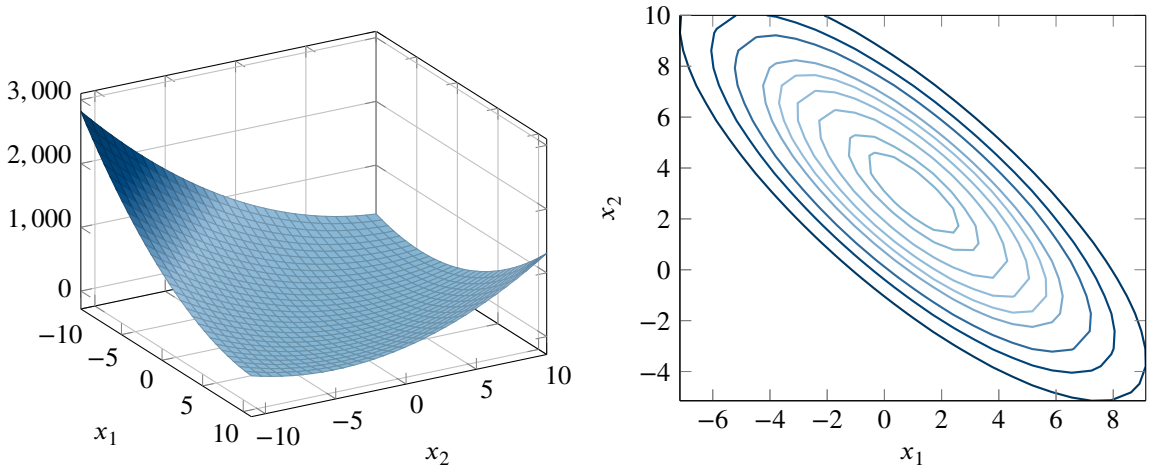
Por otro lado, **Máxima Pendiente** emplea 82 iteraciones con error 0.18034, frente a las 279 iteraciones y error 0.03903 de **Gradientes Conjugados**; aunque es más rápida, su solución se aleja más del óptimo global.

4.2 Función de Booth

La función de Booth está definida como:

$$f_1(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2.$$

Está función la evaluaremos en $x_i \in [-10, 10]$, para $i = 1, 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (1, 3)$ y al evaluarlo en la función se obtiene $f_1(\mathbf{x}^*) = 0$.



(a) Gráfica de la función de Booth

(b) Curvas de nivel de la función de Booth

Figura 4.2

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	21	(1.000000, 3.000000)	0.000000	4.4658×10^{-7}
Método de Newton	2	(1.000000, 3.000000)	0.000000	0.000000
Cuasi-Newton Rango 1	5	(1.000000, 3.000000)	0.000000	0.000000
Gradientes Conjugados	2	(1.000000, 3.000000)	0.000000	0.050401
Hooke-Jeeves	47	(1.000000, 3.000000)	0.000000	0.000000

Tabla 4.2 Resultados de la función de Booth usando *multistart* con $N = 2500$

Como se observa en la tabla, todos los métodos convergen al óptimo (1, 3) con un valor de función de cero. Para evaluar su rendimiento, usamos dos métricas: la eficiencia, medida por el número de iteraciones hasta la convergencia, y la precisión, cuantificada por el error residual, que debe tender a cero para garantizar la exactitud.

En el grupo de métodos que alcanzaron un error nulo, el **Método de Newton** destaca como el más eficiente, requiriendo únicamente **2 iteraciones** para converger. Le sigue de cerca el método **Cuasi-Newton de Rango 1**, que también logra una precisión perfecta en tan solo 5 iteraciones, consolidándose como una excelente alternativa. Aunque el método de **Hooke-Jeeves** también encuentra la solución exacta, su costo computacional es significativamente mayor, con 47 iteraciones.

Respecto a los métodos con un error residual, se presenta un interesante compromiso entre velocidad y precisión. El método de **Gradientes Conjugados** es tan rápido como el de Newton, llegando al óptimo en solo **2 iteraciones**, pero a expensas de la precisión, ya que registra el error más alto de todos. Por su parte, **Máxima Pendiente** (con Armijo) requiere 21 iteraciones y finaliza con un error muy pequeño, aunque no nulo.

En conclusión, para la función de Booth, el **Método de Newton** es indiscutiblemente superior, ofreciendo una combinación inmejorable de velocidad y exactitud. Si bien **Gradientes Conjugados** iguala su velocidad, su considerable error lo hace menos fiable. Los métodos de **Hooke-Jeeves** y, en menor medida, **Máxima Pendiente** resultan ser los que demandan un mayor esfuerzo computacional.

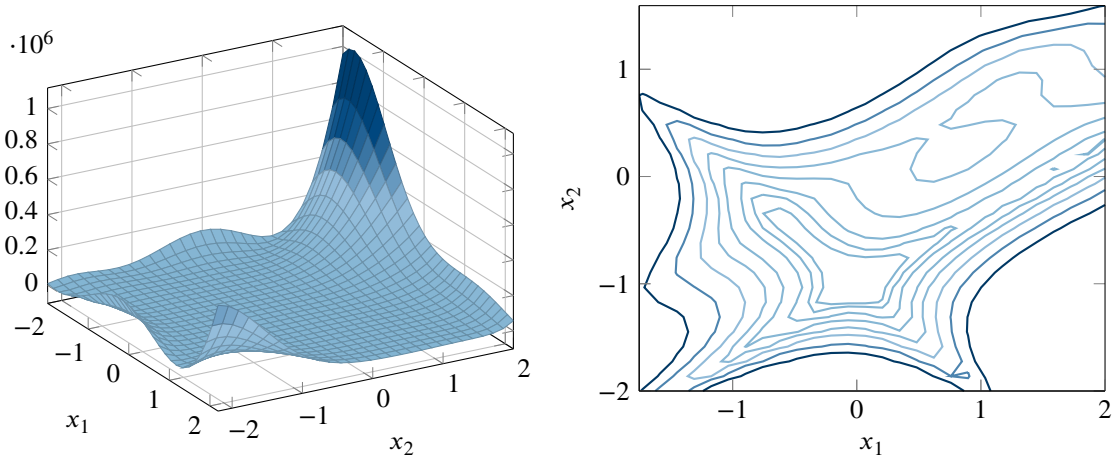
Para problemas similares a la función de Booth, conviene valorar no solo la velocidad sino también el coste de calcular la Hessiana. El Método de Newton es muy rápido y preciso, pero su necesidad de la matriz Hessiana puede volverse prohibitiva en alta dimensión. En cambio, Gradientes Conjugados y Cuasi-Newton reducen el esfuerzo por iteración al aproximar la curvatura, sacrificando algo de exactitud, y resultan más adecuados cuando no se dispone de derivadas segundas o se trabaja a gran escala.

4.3 Función Goldstein-Price

La función Goldstein-Price está definida como:

$$f(\mathbf{x}) = \left[1 + \left(x_1 + x_2 + 1\right)^2 \left(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2\right)\right] \times \left[30 + \left(2x_1 - 3x_2\right)^2 \left(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2\right)\right].$$

Está función la evaluaremos en $x_i \in [-2, 2]$, para $i = 1, 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (0, -1)$ y al evaluarlo en la función se obtiene $f(\mathbf{x}^*) = 3$.



(a) Gráfica de la función Goldstein-Price **(b)** Curvas de nivel de la función Goldstein-Price

Figura 4.3

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	11	$(3.2909 \times 10^{-10}, -1)$	3.000000	0.000000
Método de Newton	11	$(-0.000000, -1.000000)$	4.000000	0.000000
Cuasi-Newton Rango 1	11	$(-0.000000, -1.000000)$	3.000000	0.000000
Gradientes Conjugados	648	$(-0.041638, -1.010081)$	3.405280	0.235384
Hooke-Jeeves	29	$(0.000000, -1.000000)$	3.000000	0.000000

Tabla 4.3 Resultados de la función Goldstein-Price usando *multistart* con $N = 2500$

Observamos que los métodos de **Máxima Pendiente**, **Newton** y **Cuasi-Newton Rango 1** convergen en solo 11 iteraciones, mientras que **Hooke–Jeeves** requiere 29. Sin embargo, solo Máxima Pendiente y Cuasi-Newton Rango 1 alcanzan el valor correcto de la función. El Método de Newton, a pesar de localizar exactamente el óptimo, reporta $f = 4$, lo cual sugiere un posible error en la evaluación de la función.

Por el contrario, los **Gradientes Conjugados** resultan claramente ineficientes: tras 648 iteraciones no llegan al óptimo global, convergiendo a $(-0.0416, -1.0101)$ con $f \approx 3.4053$ y un error residual significativo. Este comportamiento denota sensibilidad a la forma del paisaje y un coste computacional excesivo.

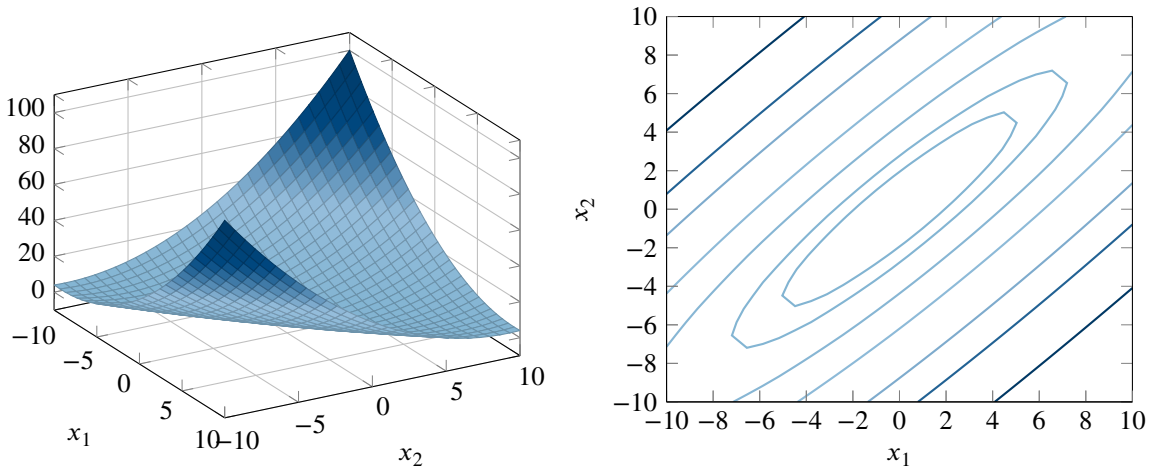
En conclusión, los métodos más robustos para Goldstein–Price son **Máxima Pendiente** y **Cuasi-Newton Rango 1**, que equilibran rapidez (11 iteraciones) y exactitud en la evaluación de la función. Hooke–Jeeves ofrece convergencia exacta pero a mayor costo iterativo, mientras que Newton, pese a su velocidad, requiere revisar su implementación de la función, y Gradientes Conjugados no resulta recomendable debido a su elevado número de iteraciones y error final.

4.4 Función de Matyas

La función de Matyas está definida como:

$$f(\mathbf{x}) = 0.26\left(x_1^2 + x_2^2\right) - 0.48x_1x_2.$$

Está función la evaluaremos en $x_i \in [-10, 10]$, para $i = 1, 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (0, 0)$ y al evaluarlo en la función se obtiene $f(\mathbf{x}^*) = 0$.



(a) Gráfica de la función de Matyas

(b) Curvas de nivel de la función de Matyas

Figura 4.4

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	289	$(-1.6979 \times 10^{-5}, -1.6979 \times 10^{-5})$	0.000000	9.7774×10^{-7}
Método de Newton	2	(0.000000, 0.000000)	0.000000	0.000000
Cuasi-Newton Rango 1	4	(−0.000000, 0.000000)	0.000000	0.000000
Gradientes Conjugados	268	(0.000000, 0.000000)	0.000000	0.373552
Hooke-Jeeves	42	(−0.000000, −0.000000)	0.000000	0.000000

Tabla 4.4 Resultados de la función de Matyas usando *multistart* con $N = 2500$

Los métodos **Método de Newton** y **Cuasi-Newton Rango 1** convergen de forma casi instantánea, en 2 y 4 iteraciones respectivamente, alcanzando exactamente (0, 0) con valor de función nulo y error cero. Esto confirma su gran eficiencia y precisión al resolver funciones cuadráticas suaves.

Por su parte, **Hooke–Jeeves** también logra converger con total exactitud, aunque requiere 42 iteraciones, lo que indica un coste computacional moderado. El método de **Máxima Pendiente** precisa 289 iteraciones para aproximarse a (0, 0), con un error residual muy bajo, reflejando su eficacia pero un mayor esfuerzo iterativo.

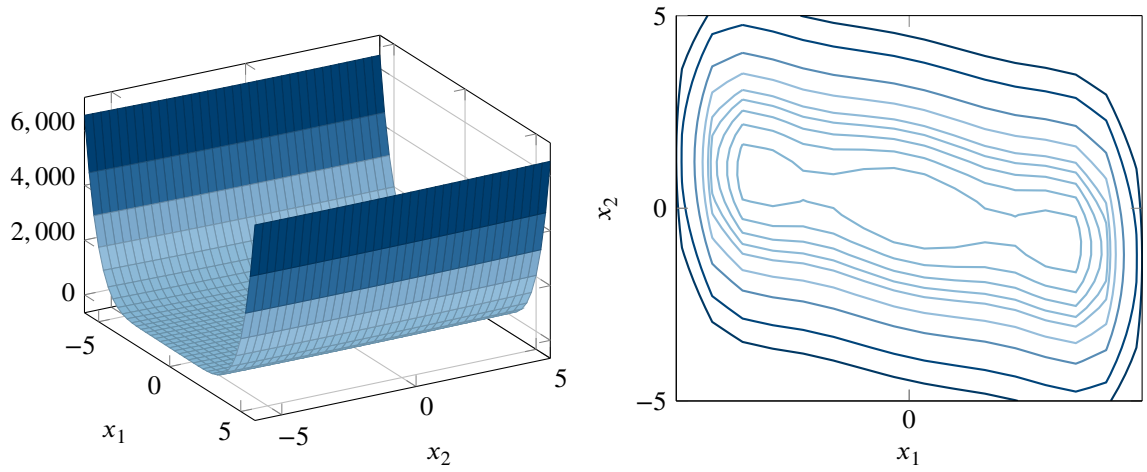
En contraste, los **Gradientes Conjugados** no consiguen situarse exactamente en el óptimo global tras 268 iteraciones, quedando en (0, 0) con un elevado error residual. Por tanto, el método de Newton y Cuasi-Newton resultan las opciones más recomendables para la función de Matyas, mientras que Gradientes Conjugados presenta un rendimiento claramente inferior.

4.5 Función Three-Hump Camel

La función de Matyas está definida como:

$$f(\mathbf{x}) = 2x_1^2 - 1.05x_1^4 + \frac{x_1^6}{6} + x_1x_2 + x_2^2.$$

Está función la evaluaremos en $x_i \in [-5, 5]$, para $i = 1, 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (0, 0)$ y al evaluarlo en la función se obtiene $f(\mathbf{x}^*) = 0$.



(a) Gráfica de la función Three-Hump Camel

(b) Curvas de nivel de la función Three-Hump Camel

Figura 4.5

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	17	(0.000000, −0.000000)	0.000000	0.000005
Método de Newton	4	(0.000000, 0.000000)	0.000000	0.000000
Cuasi-Newton Rango 1	5	(0.000000, 0.000000)	0.000000	0.000000
Gradientes Conjugados	4	(0.000000, 0.000000)	0.000000	0.062808
Hooke-Jeeves	53	(0.000000, −0.000000)	0.000000	0.000000

Tabla 4.5 Resultados de la función Three-Hump Camel usando *multistart* con $N = 2500$

Observamos que el **Método de Newton** alcanza el óptimo global (0, 0) en tan solo 4 iteraciones con error nulo, seguido muy de cerca por el **Cuasi-Newton Rango 1** en 5 iteraciones, también con precisión perfecta. El

método de **Máxima Pendiente** requiere 17 iteraciones para aproximarse a $(0,\ 0)$ con un error residual mínimo de 5×10^{-6}, lo que demuestra un buen equilibrio entre coste iterativo y exactitud.

El método de Hooke–Jeeves converge en 53 iteraciones con error cero, pero requiere más esfuerzo computacional. Los Gradientes Conjugados alcanzan $(0,\ 0)$ en 4 iteraciones, pero con un error elevado de 0.0628.

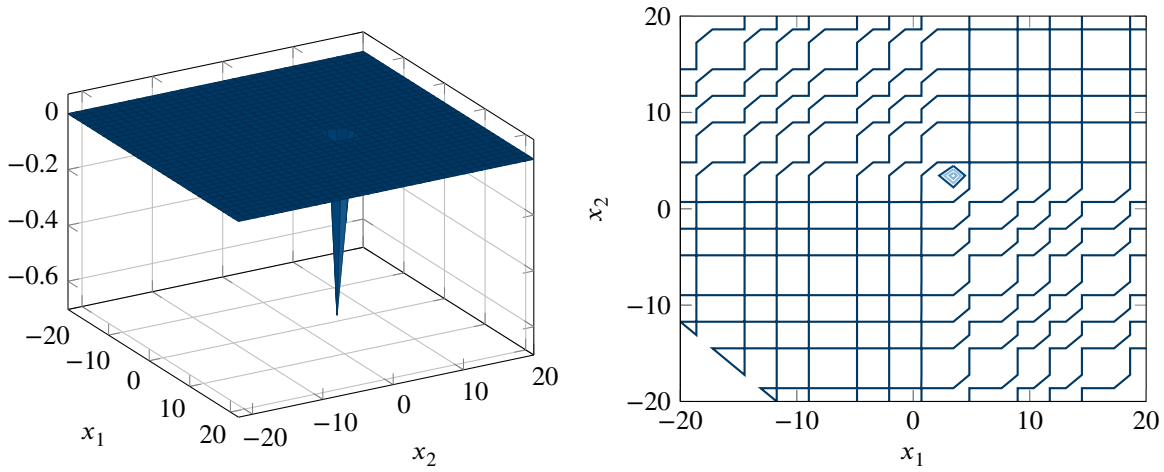
En conclusión, para la función Three-Hump Camel, Newton y Cuasi-Newton destacan por su rapidez y exactitud. Máxima Pendiente y Hooke–Jeeves ofrecen soluciones precisas a costa de más iteraciones, mientras que Gradientes Conjugados, pese a su velocidad, pierde precisión y no es recomendable.

4.6 Función de Easom

La función de Easom está definida como:

$$f(\mathbf{x}) = -\cos(x_1)\cos(x_2)e^{-(x_1-\pi)^2-(x_2-\pi)^2}.$$

Está función la evaluaremos en $x_i \in [-100,\ 100]$, para $i = 1,\ 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (\pi,\ \pi)$ y al evaluarlo en la función se obtiene $f(\mathbf{x}^*) = -1$.



(a) Gráfica de la función de Easom

(b) Curvas de nivel de la función de Easom

Figura 4.6

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	0	(−25.092, 90.1429)	0.000000	Inf
Método de Newton	4	(3.141593, 3.141593)	−1.000000	0.000000
Cuasi-Newton Rango 1	1	(3.859517, 7.948789)	−0.000000	4.860509
Gradientes Conjugados	0	(−4.160045, 19.026372)	0.000000	0.000000
Hooke-Jeeves	49	(3.141593, 3.141593)	−1.000000	0.000000

Tabla 4.6 Resultados de la función de Easom usando *multistart* con $N = 2500$

El Método de Newton alcanza el óptimo global (π, π) con $f(\pi, \pi) = -1$ en 4 iteraciones, demostrando alta eficiencia. Hooke–Jeeves también converge a (π, π) y $f = -1$, pero en 49 iteraciones, lo que incrementa el coste computacional. Cuasi-Newton Rango 1 se detiene en (3.8595, 7.9488) tras 1 iteración, con $f \approx 4.860509$. Los métodos de Máxima Pendiente y Gradientes Conjugados no progresan: el primero retorna función 0 con error infinito, mientras que el segundo queda en un punto alejado sin mejorar el valor óptimo.

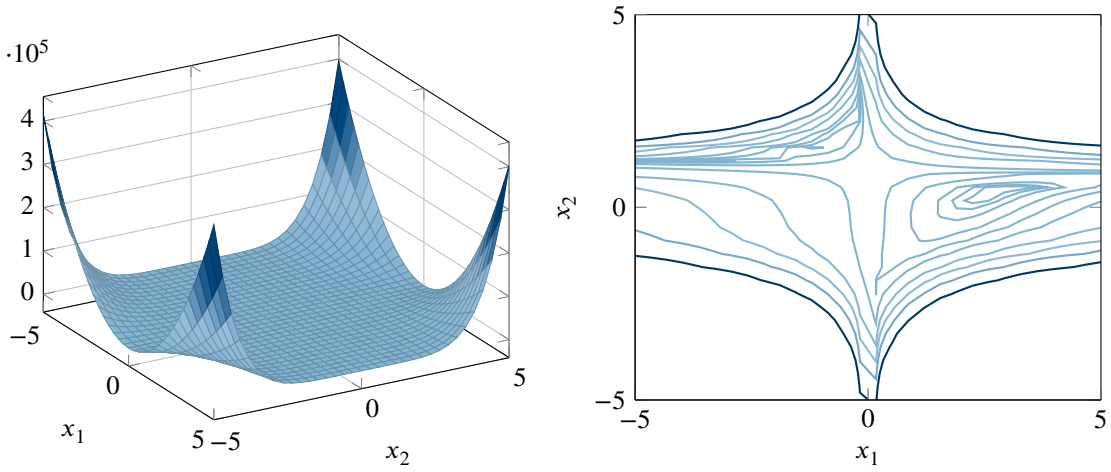
En conclusión, el Método de Newton es la mejor opción para la función de Easom, ya que combina rapidez y exactitud en la búsqueda del mínimo real. Hooke–Jeeves es fiable pero lento, Cuasi–Newton es rápido pero impreciso, y Máxima Pendiente y Gradientes Conjugados no convergen bien.

4.7 Función de Beale

La función de Beale está definida como:

$$f(\mathbf{x}) = (1.5 - x_1 + x_1 x_2)^2 + \left(2.25 - x_1 + x_1 x_2^2\right)^2 + \left(2.625 - x_1 + x_1 x_2^3\right)^2.$$

Está función la evaluaremos en $x_i \in [-4.5, 4.5]$, para $i = 1, 2$. En (Sonja Surjanovic, 2013), se menciona que el mínimo global está en $\mathbf{x}^* = (3, 0.5)$ y al evaluarlo en la función se obtiene $f(\mathbf{x}^*) = 0$.



(a) Gráfica de la función de Beale **(b)** Curvas de nivel de la función de Beale

Figura 4.7

Al correr nuestros programas, obtenemos la siguiente tabla:

Método	Iteraciones	Punto óptimo	Función evaluada	Error
Máxima Pendiente	478	(3.000000, 0.500000)	0.000000	0.000000
Método de Newton	11	(3.000000, 0.500000)	0.000000	0.000000
Cuasi-Newton Rango 1	14	(3.000000, 0.500000)	0.000000	0.000000
Gradientes Conjugados	63	(3.000000, 0.500000)	0.000000	0.048560
Hooke-Jeeves	61	(3.000000, 0.500000)	0.000000	0.000000

Tabla 4.7 Resultados de la función de Beale usando *multistart* con $N = 2500$

Los métodos **Método de Newton** y **Cuasi-Newton Rango 1** destacan por su rapidez y exactitud: convergen al óptimo global (3, 0.5) en apenas 11 y 14 iteraciones, respectivamente, logrando valor de función cero y error residual nulo.

El **Hooke–Jeeves** también alcanza la solución exacta con error cero, aunque requiere 61 iteraciones, mientras que los **Gradientes Conjugados** tardan 63 iteraciones y presentan un error residual de 0.04856, lo que revela menor fiabilidad en la convergencia.

Por último, **Máxima Pendiente** resulta el menos eficiente, necesitando 478 iteraciones para converger exactamente a (3, 0.5). En conclusión, Método de Newton y Cuasi-Newton Rango 1 son las opciones más recomendables para la función de Beale, seguidos por Hooke–Jeeves; Gradientes Conjugados y Máxima Pendiente muestran desventajas por su error o coste computacional elevado.

5
Conclusiones

En este proyecto nos permitió poder evaluara los métodos de optimización con alguna característica particular, por ejemplo el Método de Máxima Pendiente el tamaño de paso se cálculo con Armijo, ya que consideramos que era la mejor forma de aproximar por este método, en Cuasi-Newton escogimos por Rango 1, Gradientes Conjugados al calcular el beta se hizo por medio de Fletcher-Reeves, así para el resto de métodos.

Los cinco métodos de optimización (Máxima Pendiente con Armijo, Newton, Cuasi-Newton Rango 1, Gradientes Conjugados y Hooke-Jeeves), que se probaron por medio del lenguaje de programación MATLAB, a las siete funciones las cuales 6 eran multimodales y una cuadrática, nos permitió poder hacer un analices más exacto de que método nos conviene más, aplicando la técnica del multistart que nos daba 2500 puntos aleatorios en un intervalo y cada punto lo toma como el punto inicial al programar y así poder iterar hasta llegar al óptimo.

En general, se observó que el método de Newton fue consistentemente uno de los más eficientes, logrando convergencia en pocas iteraciones y alcanzando con frecuencia el mínimo global, especialmente en funciones suaves y cuadráticas como Booth, Matyas y Beale. No obstante, en funciones más complejas o altamente no lineales (como Easom o Goldstein-Price), este método mostró sensibilidad al cálculo de la Hessiana.

El método Cuasi-Newton Rango 1 demostró ser un método adversario para el Método de Newton, ya que, presenta errores casi nulos o menores a $\times 10^{-9}$ y tener una buena precisión y número de iteraciones, siendo el más eficiente para la función de Bohachevsky. Por otra parte, Hooke-Jeeves, fue un método preciso, sin embargo, era el más costoso en términos computacionales, debido a su naturaleza de búsqueda sin derivadas.

El método de Gradientes Conjugados tuvo complicaciones en funciones con múltiples óptimos, acumulando un mayor número de iteraciones y errores significativos en varios casos, como en las funciones de Easom y Matyas. En contraste, el método de Máxima Pendiente con Armijo fue más robusto, pero también más costoso en iteraciones, destacando cuando la función favorece el descenso más simple.

En conclusión podemos decir que no hay un método que sea mejor que los demás ya que depende mucho de la función de la naturaleza.Este análisis comparativo nos muestra la importancia de comprender la estructura del problema antes de elegir el método de optimización adecuado.

A

El Algoritmo de Hooke-Jeeves

A.1 Introducción

A.1.1 Optimización y la necesidad de métodos sin derivadas

La optimización es el proceso fundamental de encontrar la mejor solución posible para un problema dado, lo que implica maximizar o minimizar una función objetivo mediante el ajuste de un conjunto de variables o parámetros de entrada. Los métodos clásicos de optimización, como el descenso de gradiente y los métodos cuasi-Newton, han sido herramientas poderosas en este campo, pero dependen crucialmente de la disponibilidad y el cálculo de las derivadas (gradientes) de la función objetivo. Sin embargo, en una multitud de problemas del mundo real, esta información del gradiente no está disponible, es computacionalmente prohibitiva de obtener, o la función objetivo misma no es diferenciable. Esta limitación se manifiesta prominentemente en escenarios donde la función objetivo se evalúa a través de simulaciones computacionales complejas, donde la función puede comportarse como una "caja negra" (black box), o donde la función presenta discontinuidades, ruido o es multimodal. La optimización basada en simulación, en particular, a menudo produce funciones de costo que son aproximaciones numéricas discontinuas de una función subyacente potencialmente suave, lo que puede hacer que los algoritmos basados en gradientes fallen lejos de un mínimo. En tales casos, se requiere una clase diferente de algoritmos: los métodos de optimización sin derivadas.

A.1.2 Una visión general

Existen tres métodos de optimización sin derivadas muy populares, cada uno representando un enfoque distinto para navegar por el espacio de búsqueda sin depender de la información del gradiente.

- Algoritmo de Hooke-Jeeves (HJ): Un método clásico de búsqueda directa o búsqueda por patrones (pattern

search), desarrollado en los albores de la optimización computacional. Funciona explorando sistemáticamente el vecindario de una solución actual y extrapolando en direcciones prometedoras.

- Algoritmos Genéticos (GA): Un algoritmo evolutivo heurístico inspirado en los principios de la selección natural y la genética. Opera sobre una población de soluciones candidatas, aplicando operadores genéticos como la selección, el cruce y la mutación para evolucionar hacia mejores soluciones a lo largo de generaciones.
- Optimización por Enjambre de Partículas (PSO): Un algoritmo heurístico de inteligencia de enjambre inspirado en el comportamiento social de organismos como bandadas de pájaros o bancos de peces. Utiliza una población de partículas que ajustan sus trayectorias basándose en su propia mejor experiencia y la mejor experiencia del enjambre.

En este apéndice explicaremos el Algoritmo de Hooke-Jeeves.

A.2 El Algoritmo de Hooke-Jeeves

El algoritmo de Hooke-Jeeves (HJ), desarrollado por Robert Hooke y T. A. Jeeves en 1961, es un método pionero de búsqueda directa, también conocido como búsqueda por patrones. Pertenece a una clase de métodos de optimización que dependen únicamente de la evaluación de la función objetivo, sin requerir ni intentar calcular sus derivadas. La idea central es realizar un examen secuencial de soluciones de prueba, comparando cada una con la mejor solución encontrada hasta el momento y utilizando una estrategia basada en resultados anteriores para determinar la siguiente solución a probar. El algoritmo opera principalmente a través de dos tipos de movimientos: movimientos exploratorios y movimientos de patrón. El procedimiento del algoritmo HJ se puede describir de la siguiente manera:

1. **Inicialización:** Se comienza con un punto base x_{base} en el espacio de búsqueda y se define un tamaño de paso inicial α.
2. **Movimientos exploratorios:** Se evalúa la función objetivo probando $x_{\text{base}} + \alpha e_i$ y $x_{\text{base}} - \alpha e_i$ para cada dimensión i, donde e_i es el vector unitario. Si una de estas evaluaciones mejora el valor de la función objetivo, actualiza la coordenada correspondiente.
3. **Movimientos de patrón:** Si el movimiento exploratorio fue exitoso, extrapola un nuevo punto

$$x_p = x_{\text{new}} + \left(x_{\text{new}} - x_{\text{base}}^{\text{prev}}\right).$$

   Se realiza una evaluación exploratoria en el punto x_p para encontrar x_{pp}. Si $f(x_{pp}) < f(x_{\text{new}})$, se acepta x_{pp} como el nuevo punto base.
4. **Reducción del tamaño de paso:** Si el movimiento exploratorio no encontró mejora, reduce el tamaño de paso α multiplicándolo por un factor ρ.
5. **Terminación:** El algoritmo termina si el tamaño de paso α es menor que una tolerancia predefinida ε o después de un número máximo de iteraciones.

En pseudocódigo, tenemos que:

Entrada: x_0: punto base inicial (vector n-dimensional); α: tamaño de paso inicial;
ρ: factor de reducción de paso ($0 < \rho < 1$); ε: tolerancia; f: función objetivo a minimizar

Salida : Mejor solución encontrada x_{mejor}

$x_{\text{base}} \leftarrow x_0$
$x_{\text{prev}} \leftarrow x_0$
éxito $\leftarrow$ false
while $\alpha > \varepsilon$ **do**
  `// Movimiento exploratorio`
  $x_{\text{new}} \leftarrow x_{\text{base}}$
  for $i \leftarrow 1$ **to** n **do**
    `// Probar dirección positiva`
    $x_{\text{prueba}} \leftarrow x_{\text{new}} + \alpha \cdot e_i$
    if $f(x_{prueba}) < f(x_{new})$ **then**
      $x_{\text{new}} \leftarrow x_{\text{prueba}}$
    end
    `// Probar dirección negativa`
    $x_{\text{prueba}} \leftarrow x_{\text{new}} - \alpha \cdot e_i$
    if $f(x_{prueba}) < f(x_{new})$ **then**
      $x_{\text{new}} \leftarrow x_{\text{prueba}}$
    end
  end
  if $f(x_{new}) < f(x_{base})$ **then**
    `// Éxito en exploración: realizar movimiento de patrón`
    $x_p \leftarrow x_{\text{new}} + (x_{\text{new}} - x_{\text{prev}})$
    $x_{\text{prev}} \leftarrow x_{\text{base}}$
    $x_{\text{base}} \leftarrow x_{\text{new}}$
    `// Explorar alrededor del punto de patrón`
    $x_{\text{pp}} \leftarrow \text{Explorar}(x_p, \alpha)$
    if $f(x_{pp}) < f(x_{base})$ **then**
      $x_{\text{base}} \leftarrow x_{\text{pp}}$
    end
    éxito $\leftarrow$ true;
  else
    `// Reducir tamaño de paso si no hay mejora`
    if *éxito* = *false* **then**
      $\alpha \leftarrow \rho \cdot \alpha$
    end
    $x_{\text{prev}} \leftarrow x_{\text{base}}$
    éxito $\leftarrow$ false
  end
end
return x_{base}

A.3 Implementación en MATLAB

A continuación, presentamos la implementación del algoritmo Hooke-Jeeves en MATLAB. Se hará uso de la técnica *multistart*. Este generará 2500 puntos uniformemente distribuidos en $\mathbf{x}_i \in [a, b]$. Dicho programa tomará cada uno de estos puntos como punto inicial. El programa nos regresará: el número de iteraciones, el mejor punto óptimo encontrado, la función evaluada en ese punto óptimo encontrado y el error que existe entre el punto óptimo encontrado y el verdadero punto óptimo (que se nos proporciona en Sonja Surjanovic, 2013).

```
% Parámetros comunes
a = -10; b = 10;              % Límites del dominio
n_points = 2500;              % Número de puntos iniciales
dim = 2;                      % Dimensión del problema
true_opt = [1, 3];            % Óptimo verdadero

% Generar 2500 puntos iniciales uniformemente distribuidos
rng(1); % Fijar semilla para reproducibilidad
initial_points = a + (b-a)*rand(n_points, dim);

% Definir la función objetivo
objective_func = @(x) (x(1) + 2*x(2) - 7)^2 + (2*x(1) + x(2) - ...
    5)^2; % Función de la sección 4.2

% Llamar a los optimizadores
[opt_hj, fval_hj, iter_hj, err_hj] = ...
    hooke_jeeves_ms(objective_func, initial_points, a, b, true_opt);

% Mostrar resultados
fprintf('Hooke-Jeeves:\n  Punto óptimo: [%.6f, %.6f]\n  Valor ...
    función: %.6f\n  Iteraciones: %d\n  Error: %.6f\n\n',...
         opt_hj(1), opt_hj(2), fval_hj, iter_hj, err_hj);

% Hooke-Jeeves con MultiStart
function [best_x, best_fval, best_iter, best_err] = ...
    hooke_jeeves_ms(f, initial_points, a, b, true_opt)
     % Parámetros del algoritmo
     alpha0 = 1.0;        % Tamaño de paso inicial
     rho = 0.5;           % Factor de reducción de paso
     epsilon = 1e-6;      % Tolerancia de terminación
     max_iter = 1000;     % Máximo de iteraciones por ejecución

     n_points = size(initial_points, 1);
     best_fval = inf;
     best_x = [];
     best_iter = 0;  % Almacena iteraciones de la mejor ejecución

     for i = 1:n_points
```

```
35          [x_opt, fval, iter] = hooke_jeeves(f, ...
                initial_points(i,:), alpha0, rho, epsilon, max_iter, ...
                a, b);
36
37          % Guardar iteraciones SOLO si es la mejor ejecución
38          if fval < best_fval
39              best_fval = fval;
40              best_x = x_opt;
41              best_iter = iter;  % Guarda iteraciones de esta ejecución
42          end
43      end
44
45      best_err = norm(best_x - true_opt);
46  end
47
48  function [x_opt, fval, iter] = hooke_jeeves(f, x0, alpha, rho, ...
        epsilon, max_iter, a, b)
49      x_base = x0;
50      x_prev = x0;
51      success = false;
52      iter = 0;
53      n = length(x0);
54
55      while alpha > epsilon && iter < max_iter
56          iter = iter + 1;
57          % Movimiento exploratorio
58          x_new = x_base;
59
60          for i = 1:n
61              % Prueba en dirección positiva
62              x_temp = x_new;
63              x_temp(i) = min(x_temp(i) + alpha, b);
64              if f(x_temp) < f(x_new)
65                  x_new = x_temp;
66              else
67                  % Prueba en dirección negativa
68                  x_temp = x_new;
69                  x_temp(i) = max(x_temp(i) - alpha, a);
```

```
                if f(x_temp) < f(x_new)
                    x_new = x_temp;
                end
            end
        end

        % Verificar éxito en la exploración
        if f(x_new) < f(x_base)
            % Movimiento de patrón
            x_pattern = 2*x_new - x_prev;
            % Asegurar que está dentro de los límites
            x_pattern = max(min(x_pattern, b), a);

            % Explorar alrededor del punto de patrón
            x_pp = x_pattern;
            for i = 1:n
                % Prueba en dirección positiva
                x_temp = x_pp;
                x_temp(i) = min(x_temp(i) + alpha, b);
                if f(x_temp) < f(x_pp)
                    x_pp = x_temp;
                else
                    % Prueba en dirección negativa
                    x_temp = x_pp;
                    x_temp(i) = max(x_temp(i) - alpha, a);
                    if f(x_temp) < f(x_pp)
                        x_pp = x_temp;
                    end
                end
            end

            % Actualizar puntos
            x_prev = x_base;
            if f(x_pp) < f(x_new)
                x_base = x_pp;
            else
                x_base = x_new;
            end
```

```
108                 success = true;
109             else
110                 % Reducir tamaño de paso si no hay mejora
111                 if ~success
112                     alpha = rho * alpha;
113                 end
114                 x_prev = x_base;
115                 success = false;
116             end
117         end
118
119         x_opt = x_base;
120         fval = f(x_opt);
121     end
```

Al compilar, obtenemos la siguente salida

Command Window

```
>> HJs
Hooke-Jeeves:
  Punto óptimo: [1.000000, 3.000000]
  Valor función: 0.000000
  Iteraciones: 47
  Error: 0.000000
```

Bibliografía

Alam, Tanweer, S. Qamar y Mohamed Benaida (2020). «Genetic Algorithm: Reviews, Implementations, and Applications». En: *International Journal of Engineering Pedagogy (iJEP)* 10.6, págs. 55-76. DOI: 10.3991/ijep.v10i6.14565. URL: https://online-journals.org/index.php/i-jep/article/view/14565.

Copilotly (2024). *Hooke-Jeeves Algorithm*. URL: https://www.copilotly.com/ai-glossary/hooke-jeeves-algorithm (visitado 17-05-2024).

Deepgram (abr. de 2025). *AI Glossary: Hooke-Jeeves Algorithm*. URL: https://deepgram.com/ai-glossary/hooke-jeeves-algorithm (visitado 15-05-2024).

Harder, Douglas (2021). *Hooke-Jeeves Method Example*. El contenido del vídeo describe el método Hooke-Jeeves. URL: https://www.youtube.com/watch?v=-w4ZbzetyeI (visitado 15-05-2024).

Roberts, Julia (2015). *Optimization of Metal Cutting Process Using Hook's and Jeeves Method*. Authors not clearly specified in PDF metadata or text. URL: https://repo.ijiert.org/index.php/ijiert/article/download/266/246/494.

Sonja Surjanovic Derek Bingham, Simon Fraser University (2013). *Virtual Library of Simulation Experiments: Test Functions and Datasets*. Accessed: 2025-04-10. URL: https://www.sfu.ca/~ssurjano/optimization.html.

Stanford University, SISL K-12 Outreach (2013). *Minimum of a 3D Function with Hooke-Jeeves*. Educational Material PDF. URL: https://web.stanford.edu/group/sisl/k12/optimization/MO-unit2-pdfs/2.11minimum3D2hooke-jeeves.pdf (visitado 22-05-2024).