Inteligência Artificial

2023/2024

Trabalho Prático 2: Métodos de Pesquisa

Eduardo Solteiro Pires / Paulo Moura Oliveira

Autores:

Guilherme Barros, 74397

Patrício Simões, 73407

1. Índice de Conteúdo

3. Introdução 4
4. Subida da Colina (*Hill Climbing*) 5
4.1 Construção do Algoritmo 5
4.2 Exemplos de Execução 7
4.3 Conclusões 8
5. Recozimento Simulado (*Simulated Annealing*) 10
5.1 Construção do Algoritmo 10
5.2 Exemplos de Execução 13
5.3 Conclusões 14
6. Algoritmo Genético (*Genetic Algorithm*) 15
6.1 Construção do Algoritmo 15
6.2 Exemplo de Execução 17
6.3 Conclusões 19
7. Considerações Finais 20
8. Anexos 22
8.1 Hill Cimbing (HC) 22
8.2 Simulated Annealing (SA) 23
8.2 Algoritmo Genético (AG) 25

2. Índice de Ilustrações

Figura 1 - Exemplo da Subida da Colina 6
Figura 2 - Exemplo da Evolução de x 7
Figura 3 - Exemplo de execução do SA 12
Figura 4 - Valores de crescimento do SA 12
Figura 5 - Inicialização do AG 17
Figura 6 - Geração nº23 do exemplo em questão 18
Figura 7 - Geração e valores finais do AG 18

3. Introdução

No âmbito da Unidade Curricular de Inteligência Artificial (IA), lecionada a Engenharia Informática no ano letivo de 2023/2024, o seguinte trabalho tenciona explorar e analisar três algoritmos fundamentais na área de otimização: a "Subida da Colina" (*Hill Climbing*), o "Recozimento Simulado" (*Simulated Annealing*) e o Algoritmo Genético (*Genetic Algorithm*).

O objetivo desta investigação é promover a aquisição de conhecimentos e o desenvolvimento de competências em relação a estas técnicas de busca, que desempenham um papel essencial na resolução de problemas complexos. O entendimento e aplicação destes algoritmos, permite-nos adquirir uma compreensão mais profunda de como a inteligência artificial pode ser utilizada para encontrar soluções ótimas em cenários variados e desafiadores.

Ao longo deste documento iremos descrever o funcionamento de cada algoritmo, bem como as suas construções com recurso à linguagem e ferramenta "*MATLAB",* e iremos apresentar algumas situações práticas de pesquisa e estabelecer uma comparação entre os algoritmos.

4. Subida da Colina (*Hill Climbing*)

O algoritmo da "Subida da Colina", (*Hill Climbing*) é uma técnica de busca local que visa encontrar o máximo (ou mínimo, mediante o procurado) local em um espaço de busca.

O algoritmo avalia os vizinhos diretos de um determinado ponto e move-se para o vizinho que apresenta o melhor desempenho em relação ao critério de otimização definido. Esse processo é repetido iterativamente até que não seja possível encontrar um vizinho com um desempenho superior, ou até que se atinja um critério de paragem. No entanto, a "Subida da Colina" tem suas limitações, pois pode ficar presa em máximos locais e não explorar todo o espaço de busca.

4.1 Construção do Algoritmo

Antes de proceder à construção do algoritmo em si, é importante referir que o espaço de busca teve um domínio de intervalo [0 ; 1] e que a função utilizada para pesquisa foi a seguinte:

$$f_1(x) = 4(sin(5\,\pi\,x + 0.5)^6 exp(\log_2(x - 0.8)^2),$$

Começamos então a partir de um ponto gerado, **x_current**, de forma aleatória no espaço de busca através da função **rand** que gera um número decimal entre 0 e 1.

De seguida, estabelecemos que o número máximo de iterações, **t_max**, que seria utilizado como critério de paragem, teria um valor de 400 e que o intervalo de pesquisa, **delta**, ou seja, a vizinhança de procura relativa ao ponto x_current, teria um valor de **0.005**, (1/200).

Posto isto, procedemos à implementação do algoritmo em si, que se encontra representada na seguinte figura e pode ser consultada em anexo no final deste documento:

```
% Algorítmo de pesquisa na vizinhança.

while(t <= t_max)
    x_new = x_current + delta * (2 * rand -1); % Novo ponto gerado e ponto de comparação com o anterior.
    if(fx(x_current) < fx(x_new))
        x_current = x_new; % Troca a posição do x atual para o novo x.
        data.x = x_current;
        data.time = t;
        x_array{j} = data; % Guarda a posição do x no array de evolução.
        j = j + 1;
    end
    plot(x_current, fx(x_current),'*r');
    t = t + 1;
end
```

Este algoritmo está desenhado para ser executado 400 vezes, sendo que, com cada iteração, o algoritmo tenta perceber se o novo ponto possui uma imagem maior do que o ponto atual, ou seja, tenta encontrar um ponto maximizante com cada iteração. No caso de encontrar, substitui o ponto atual pelo novo ponto e guarda registo desta iteração na estrutura **data** de modo que seja possível construir o gráfico de evolução de x em função do tempo, que pode ser consultado na Figura 2.

Em adição a isto, com cada iteração, independentemente do novo ponto ser maximizante ou não, é desenhada a sua posição no gráfico como no seguinte exemplo:

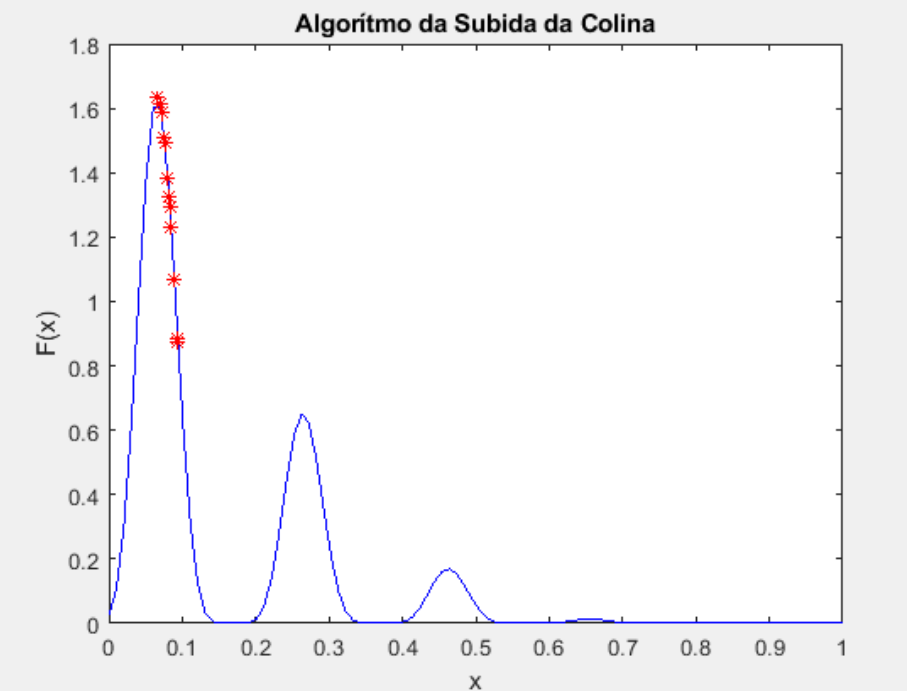


Figura 1 - Exemplo da Subida da Colina

Já a evolução do x é representada conforme o seguinte gráfico:

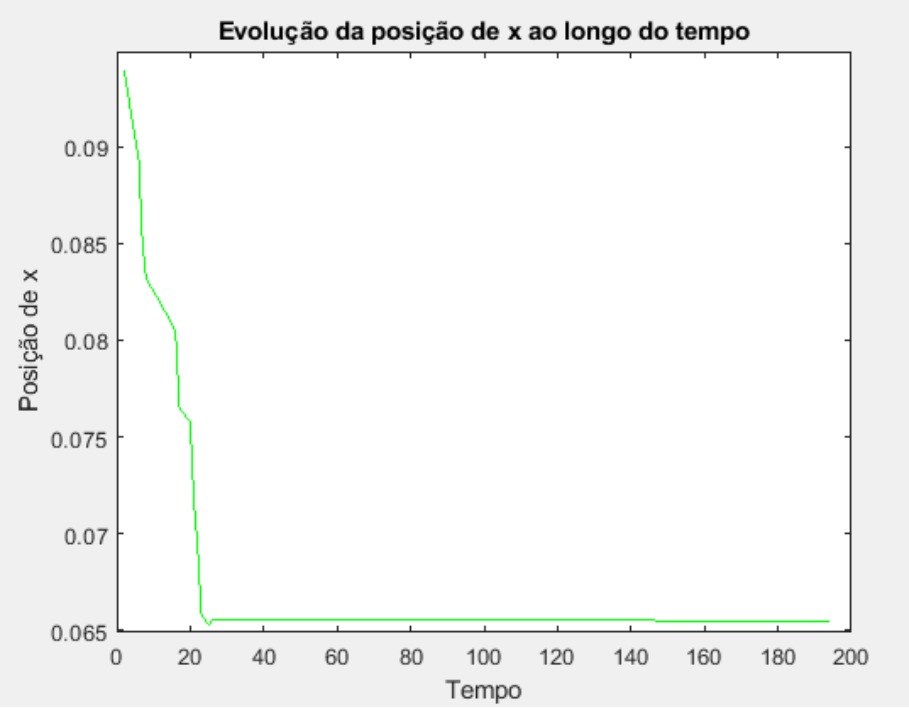
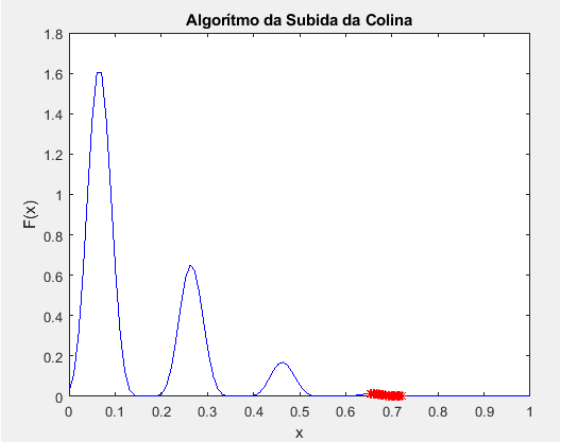
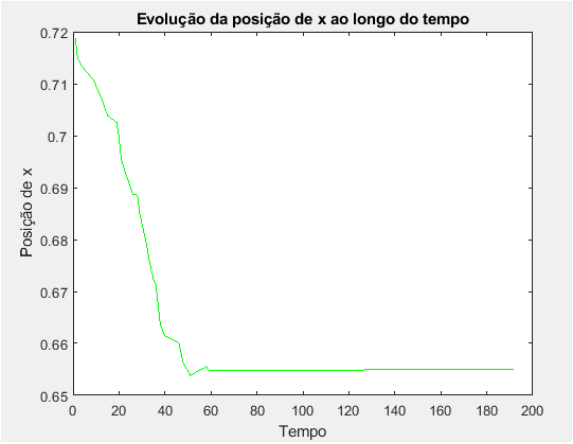


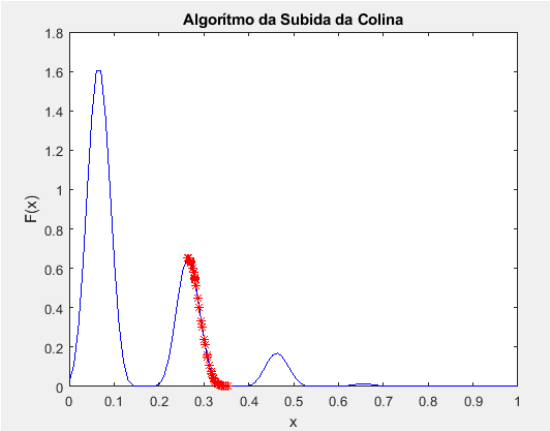
Figura 2 - Exemplo da Evolução de x

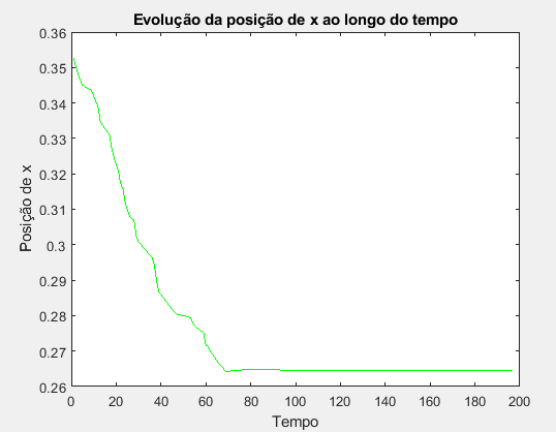
4.2 Exemplos de Execução

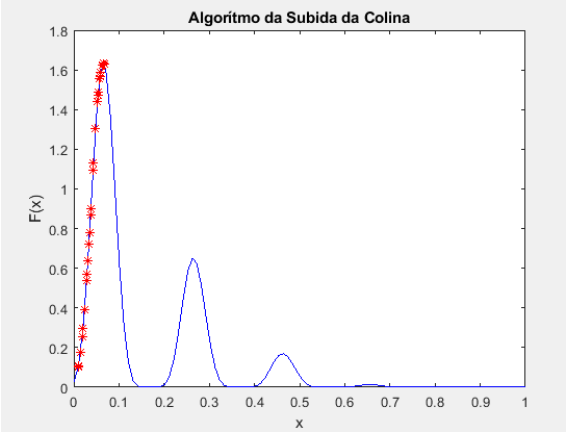
Abaixo encontram-se alguns exemplos de execução do algoritmo apresentado:

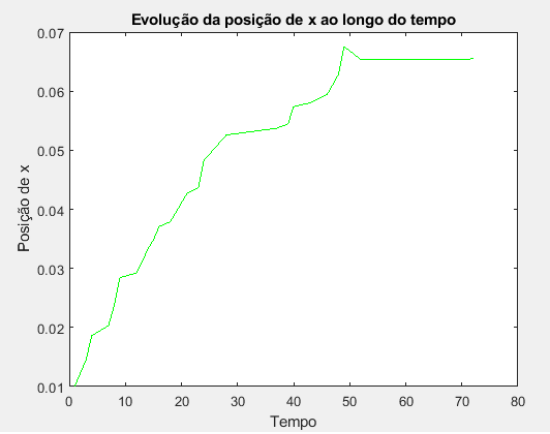












4.3 Conclusões

Depois da análise dos exemplos acima expostos, podemos tirar as seguintes conclusões:

- O Algoritmo é um algoritmo de busca local e, como tal, tem uma forte tendência a convergir para o máximo local mais próximo do ponto inicial. Isso pode ser tanto uma vantagem quanto uma desvantagem, dependendo do espaço de pesquisa do problema.
- O resultado do HC é altamente dependente do ponto inicial. Diferentes pontos iniciais podem levar a diferentes máximos locais, o que pode ser usado para explorar o espaço de pesquisa do problema executando o algoritmo várias vezes com diferentes inícios.
- Ao contrário de algoritmos como Simulated Annealing ou algoritmos genéticos, o Hill Cimbing padrão não possui mecanismos para escapar de máximos locais, limita a sua eficácia em espaços de busca

complexos. Para aprimorar esta limitação, poderíamos implementar o algoritmo com reinicialização múltipla, no entanto, isto foge ao pedido neste trabalho.

5. Recozimento Simulado (*Simulated Annealing*)

Tal como o algoritmo da “Subida de Colina”, o algoritmo de "Recozimento Simulado" (*Simulated Annealing*) é também uma técnica de busca local que visa encontrar o máximo (ou mínimo, mediante o procurado) local em um espaço de busca.

Operando à base de uma comparação do valor atual de um determinado ponto com os seus vizinhos diretos, o algoritmo faz uso de um valor mutável denominado de T (temperatura), e da diferença de valor entre o ponto atual e os seus vizinhos, para determinar para qual posição se deslocar. O que categoriza este algoritmo, é o facto do mesmo fazer uso de uma função p, regulada pelo valor T, que determina a probabilidade de o mesmo escolher entre o melhor e o pior caso possível como próximo ponto. De forma a gerir a aleatoriedade do algoritmo quando em relação à seleção do próximo ponto, o valor T é iniciado com o seu valor máximo, sendo depois reduzido ao longo das iterações de forma a diminuir a chance do pior valor ser selecionado. Isto permite uma aleatoriedade máxima no início da execução que vai diminuindo ao longo da mesma até ser nulificada, permitindo assim uma maior chance de o algoritmo não ficar preso em máximos locais e não explorar todo o espaço de busca.

5.1 Construção do Algoritmo

Antes de mais, é importante referir que o espaço de busca teve um domínio de intervalo [0,1], e que a função utilizada para a pesquisa foi a seguinte:

$$f_1(x) = 4(sin(5\,\pi\,x + 0.5)^6 exp(\log_2(x - 0.8)^2),$$

Tal como no algoritmo anterior, começamos com a implementação da geração de um ponto aleatório no espaço de busca, **x_current**, que faz uso da função **rand** para gerar um número decimal entre 0 e 1.

Após isto, foi estabelecido o número máximo de iterações, **t_max**, que é usado como critério de paragem, com um valor imutável de 400, o intervalo de pesquisa, **delta**, que representa a vizinhança de procura relativa ao ponto **x_current**, com um valor de 0.025 (1/40), o valor inicial, e máximo, da temperatura, **T**, que é usada para regular a função p e possui um valor de 90, e finalmente, o valor de decaimento da temperatura, **alfa**, com um valor de 0.94, que é usado para atualizar o novo valor de **T** a cada espaço de repetições definido, sendo este espaço regulado por **nRep**, com um valor de 10.

Com os devidos valores de importância estabelecidos, procedemos à implementação do algoritmo em si, que se encontra representada na seguinte figura e pode ser consultada em anexo no final deste documento:

```
% Algorítmo de pesquisa na vizinhança.
while(t <= t_max)
    rep = 1; % Contador de repetições.
    while(rep <= nRep)
        x_new = x_current + delta * (2 * rand -1); % Novo ponto gerado e ponto de comparação com o anterior.
        % Garante que o novo ponto gerado se encontra dentro do domínio.
        if(x_new >= 0 && x_new <= 1)
            dE = fx(x_new) - fx(x_current); % Gradiente de energia.
            p = 1/(1+exp(abs(dE)/T)); % Probabilidade de aceitar um valor pior.
            % Maximização.
            if(dE > 0)
                x_current = x_new;
                imgx = fx(x_new);
            % Caso em que aceita um valor pior, (Minimização).
            elseif(rand < p)
                x_current = x_new;
                imgx = fx(x_new);
            end
            % Guarda os valores numa estrutura para desenhar os gráfico.
            data.x = x_current;
            data.y = fx(x_current);
            data.de = dE;
            data.t = T;
            data.p = p;
            values{i} = data;
            i = i + 1;
            rep = rep + 1;
        end
    end
    T = T * alfa; % Diminuição da temperatura.
    t = t + 1;
end
```

O algoritmo desenvolvido foi concebido para ser executado 400 vezes, fazendo uso do gradiente de energia dE, e da função p, a cada iteração, para determinar a chance de escolha do melhor ou pior caso disponível na vizinhança do ponto atual. Dependendo do valor de p, o algoritmo escolhe o melhor ou pior caso como novo ponto atual, fazendo depois uma gestão da temperatura, com base na diferença entre o valor de nRep e o valor do contador local de repetições.

Adicionalmente, o algoritmo guarda também registo de múltiplos valores da iteração atual na estrutura **data,** de modo a que seja possível construir gráficos

de monitorização de evolução dos valores x, y, dE, T, p e x_current. Segue-se agora um exemplo dos gráficos anteriormente referidos:

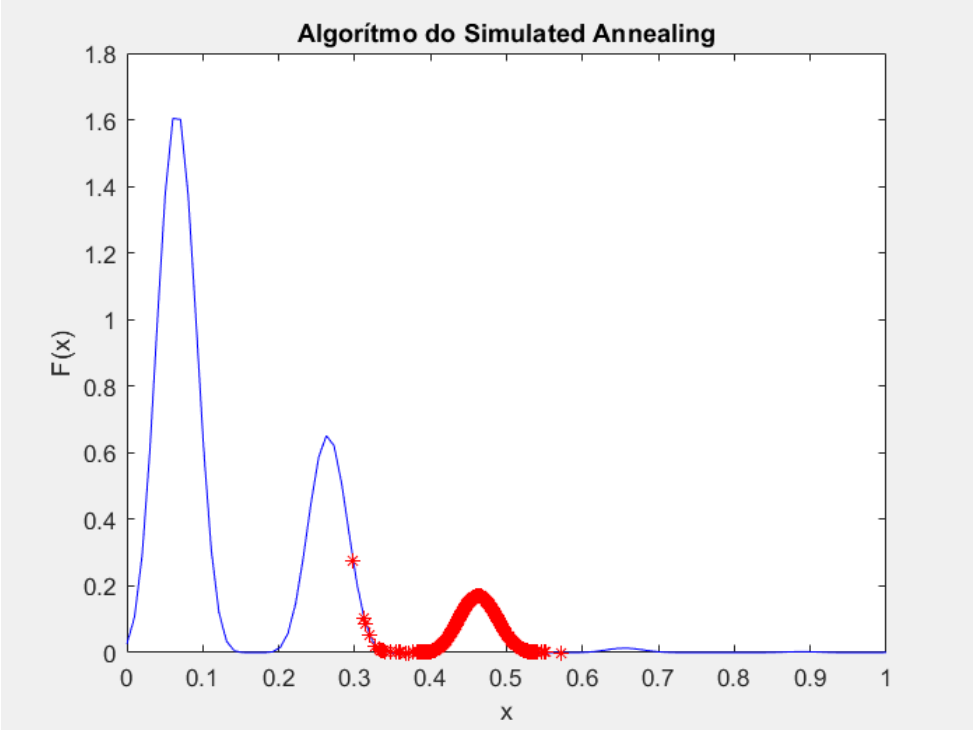


Figura 3 - Exemplo de execução do SA

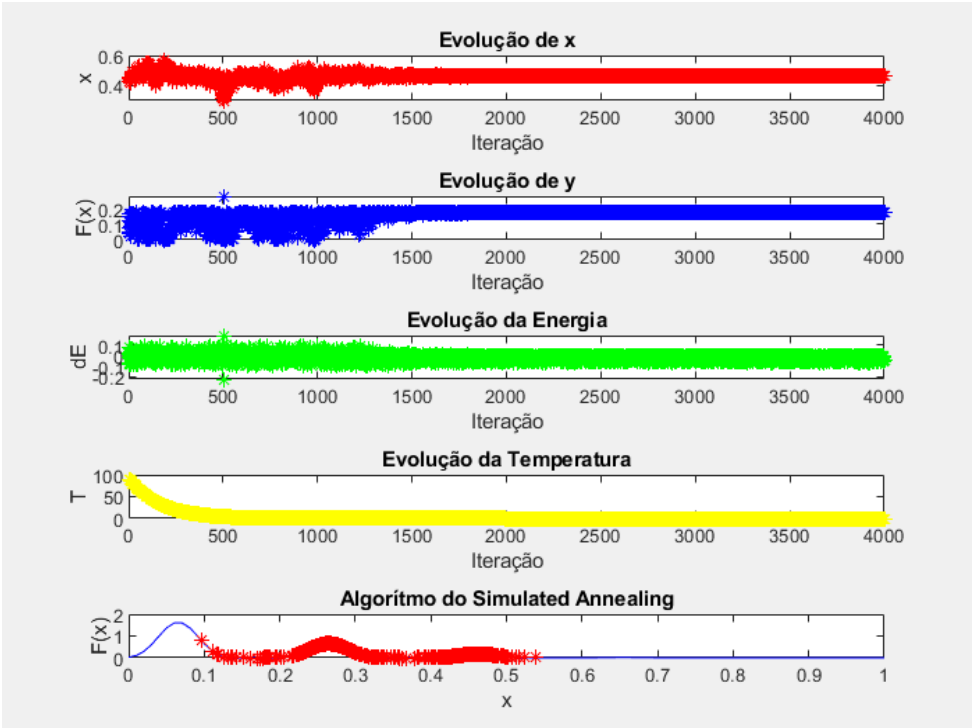
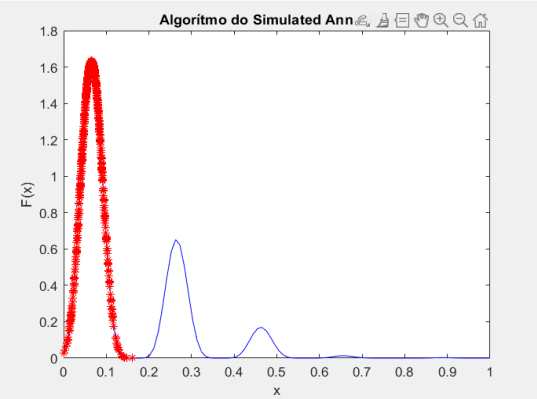
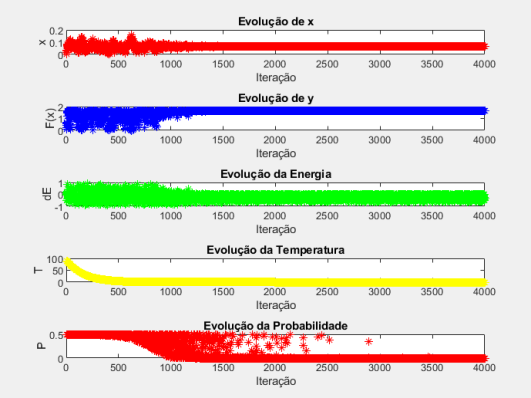


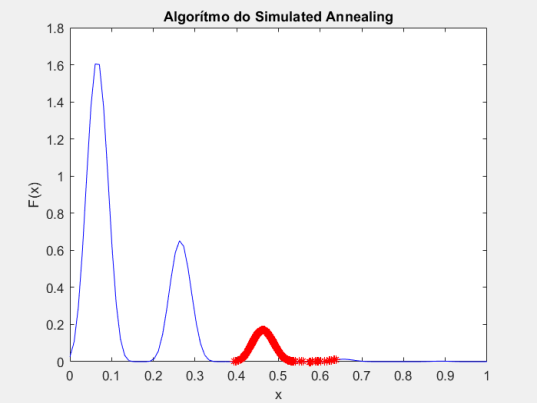
Figura 4 - Valores de crescimento do SA

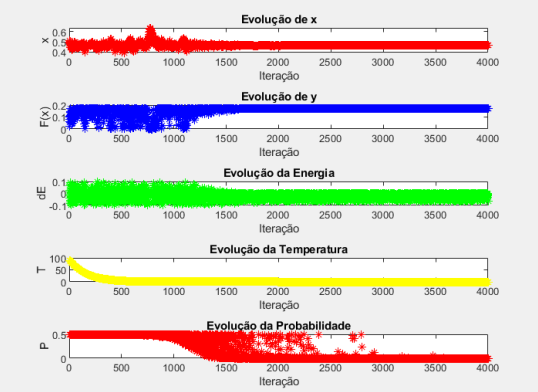
5.2 Exemplos de Execução

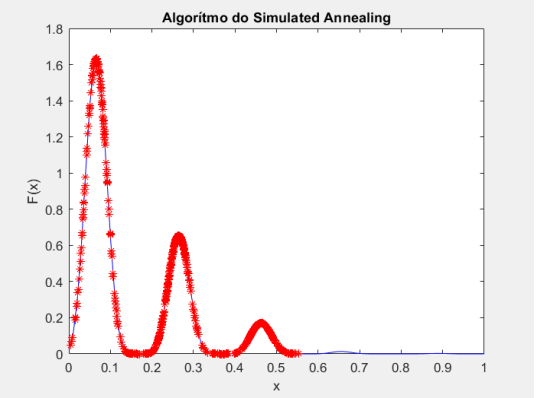
Abaixo encontram-se alguns exemplos de execução do algoritmo apresentado:

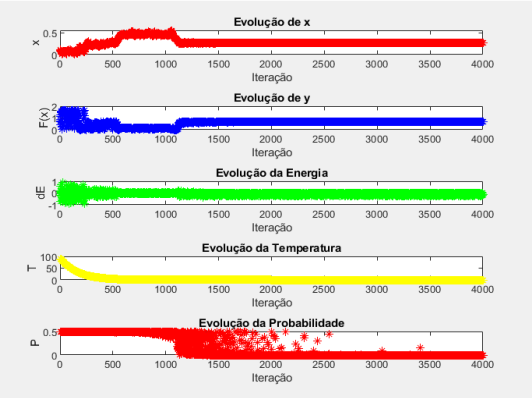












5.3 Conclusões

Depois da análise dos exemplos acima expostos, podemos tirar as seguintes conclusões:

- O algoritmo do SA é projetado para permitir movimentos temporários para soluções de pior qualidade, o que pode ajudar a evitar que o algoritmo fique preso em máximos locais, aumentando as chances de encontrar um máximo global.
- A taxa na qual a temperatura é reduzida é crítica para o sucesso do SA. Uma redução muito rápida pode levar a uma convergência prematura, enquanto uma redução muito lenta pode tornar o algoritmo ineficientemente longo.
- Nas fases iniciais, com temperaturas mais altas, o SA favorece a exploração do espaço de busca, aceitando soluções de pior qualidade. Conforme a temperatura diminui, o algoritmo torna-se mais refinado, favorecendo exploração em torno de áreas promissoras.
- Diferentes execuções do SA podem levar a resultados distintos devido à sua natureza estocástica.

6. Algoritmo Genético (*Genetic Algorithm*)

Algoritmos genéticos, (AGs), são métodos de busca e otimização inspirados no processo de seleção natural e na genética. São particularmente eficazes para resolver problemas complexos onde as soluções podem ser representadas por um conjunto de parâmetros, ou "genes", e a qualidade de uma solução pode ser avaliada por uma função de "aptidão".

Estes operam com uma "população" de soluções iniciais, e, a cada "geração", aplicam operações inspiradas seleção natural e as suas características, como cruzamentos e mutações, para criar uma nova geração de soluções. A seleção favorece as soluções mais aptas para reprodução, enquanto o cruzamento e a mutação introduzem variações que podem levar a novas soluções com aptidão ainda maior. Ao longo das gerações, a população "evolui" em direção a soluções cada vez melhores.

6.1 Construção do Algoritmo

O algoritmo genético que desenvolvemos segue esta abordagem clássica, mas com algumas particularidades. Inicializamos o nosso algoritmo com uma população de soluções codificadas como cromossomas binários, onde cada bit pode ser considerado um gene:

```
% Função que inicializa a populaçao
function CHROME = init_pop(pop_size, lchrome)
    % CHROME - Matriz da população atual
    CHROME = randi([0, 1], pop_size, lchrome);
end
```

A aptidão de cada cromossoma é determinada por uma função objetivo específica do problema em questão, (a mesma função utilizada nos algoritmos anteriores):

```
% Função objetivo.
function y = objective_function(x)
    y = 4 * (sin(5 * pi * x + 0.5).^6) .* exp(log2((x - 0.8).^2));
end
```

Utilizamos uma função de “seleção por roleta” para escolher os pais para a próxima geração, garantindo que cromossomas com maior aptidão tenham mais chances de serem selecionados, embora aqueles com menor aptidão ainda possam ser escolhidos, preservando a diversidade genética:

```
% Seleciona os indivíduos pais da próxima geração com base na aptidão de cada indivíduo.
function index = roulette_selection(POP)
    cumulative_sum = cumsum(POP);
    roulette_pick = rand * cumulative_sum(end);
    index = find(cumulative_sum >= roulette_pick, 1, 'first');
end
```

Após a seleção, aplicamos um cruzamento de um ponto, onde um ponto de corte é escolhido aleatoriamente, e partes dos cromossomas dos pais são trocadas para formar dois novos cromossomas, ou "filhos":

```
% Realiza um cruzamento de um ponto entre dois pais para criar dois filhos.
% O ponto de cruzamento é escolhido aleatoriamente.
function [child1, child2] = single_point_crossover(parent1, parent2, p_cross)
    if rand <= p_cross
        % Verifica se o cruzamento ocorrerá, com base na probabilidade de cruzamento.
        point = randi(length(parent1) - 1);
        child1 = [parent1(1:point), parent2(point+1:end)];
        child2 = [parent2(1:point), parent1(point+1:end)];
    else
        % Se não houver cruzamento, os filhos são cópias dos pais.
        child1 = parent1;
        child2 = parent2;
    end
end
```

Além disso, incorporamos a mutação, uma pequena alteração aleatória nos genes, para explorar novas regiões do espaço de busca e evitar a convergência prematura para máximos locais.

```
% Aplica mutação a um indivíduo (cromossoma).
function individual = mutate(individual, p_mutation)
    for i = 1:length(individual)
        % Verifica se a mutação ocorrerá para cada gene.
        if rand <= p_mutation
            individual(i) = 1 - individual(i);
        end
    end
end
```

A cada geração, avaliamos a nova população com a função objetivo e selecionamos as melhores soluções. Este processo é repetido por um número predeterminado de gerações ou até que um critério de convergência seja atingido:

```
% Calcula a aptidão da população.
function [POP, POP_x] = evaluate_population(CHROME, lchrome, limits)
    pop_size = size(CHROME, 1); % Número de indivíduos na população.
    POP = zeros(1, pop_size); % Vetor de aptidão.
    POP_x = zeros(1, pop_size); % Vetor de valores da variável de decisão.

    % Itera sobre a população para calcular a aptidão de cada cromossoma.
    for i = 1:pop_size
        % Converte o cromossoma binário em valor real dentro dos limites.
        x = bin2dec(num2str(CHROME(i, :))) / (2^lchrome - 1);
        x = x * (limits(2) - limits(1)) + limits(1);
        POP_x(i) = x; % Armazena o valor real da variável de decisão.
        POP(i) = objective_function(x); % Calcula e armazena a aptidão.
    end
end
```

Ao longo do algoritmo, mantemos registos da melhor solução encontrada e da média de aptidão da população, o que nos permite monitorar o progresso e ajustar os parâmetros, se necessário.

6.2 Exemplo de Execução

O algoritmo começa por desenhar o mesmo gráfico em que os algoritmos anteriores operavam. (O espaço de pesquisa):

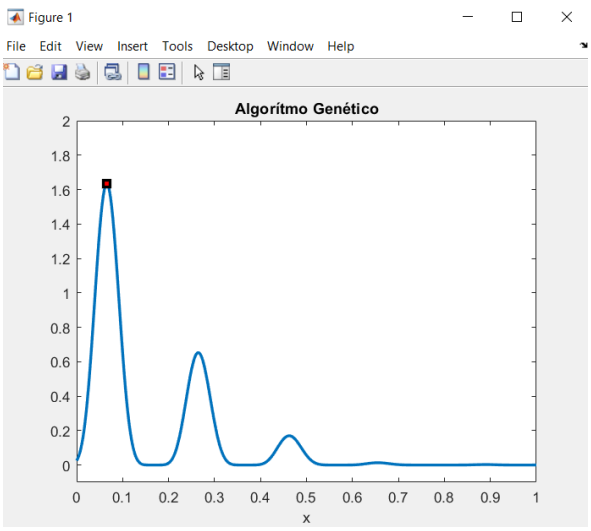


Figura 5 - Inicialização do AG

De seguida, este procede à iteração entre gerações, (80 gerações no caso), onde podemos observar os vários cromossomas da geração atual:

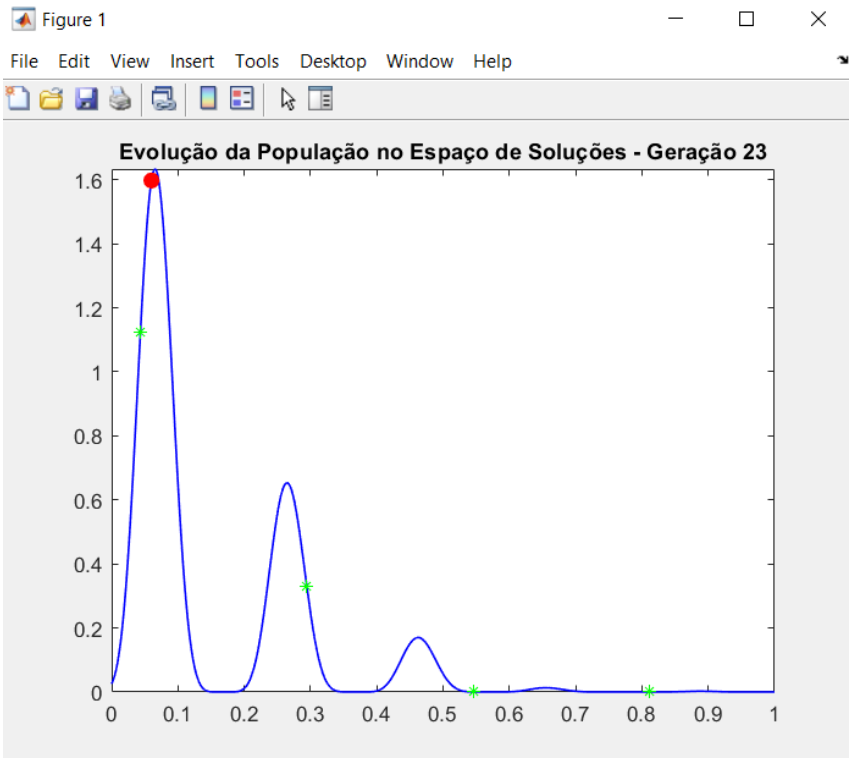


Figura 6 - Geração nº23 do exemplo em questão

Por fim, é apresentada a geração final, em conjunto com os gráficos de melhor aptidão por geração e da aptidão média por geração:

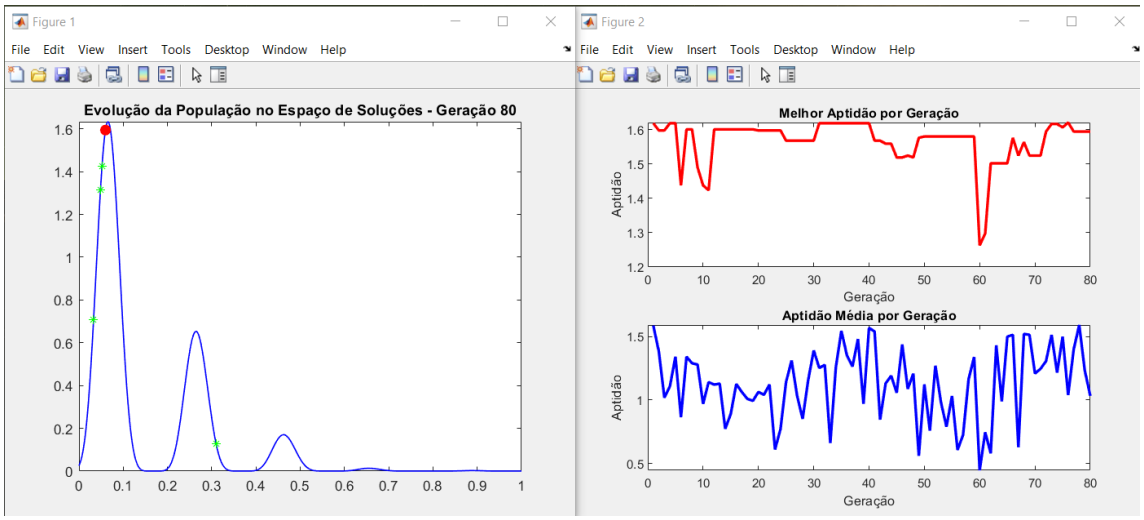


Figura 7 - Geração e valores finais do AG

6.3 Conclusões

Depois da execução do algoritmo genético, podemos tirar as seguintes conclusões:

- O algoritmo não nos garante que o máximo encontrado é um máximo global;
- O algoritmo pode explorar eficientemente grandes espaços de busca, identificando regiões de alta aptidão sem a necessidade de uma busca exaustiva.
- A diversidade da população inicial e as operações de cruzamento e mutação ajudam a evitar que o algoritmo fique preso em máximos locais.
- Ao longo das gerações, a população tende a convergir para soluções que maximizam a função de aptidão, o que demonstra a capacidade do algoritmo de adaptar e refinar soluções. Esta convergência, no entanto, não nos garante que a solução seja ótima.
- Ajustar as taxas de cruzamento e mutação é crucial; taxas muito altas podem levar a uma exploração excessiva e falta de refinamento, enquanto taxas muito baixas podem não introduzir variação suficiente, levando a uma convergência prematura.

7. Considerações Finais

A execução e comparação dos algoritmos de Hill Climbing (HC), Simulated Annealing (SA) e Algoritmos Genéticos (AGs) fornece uma visão abrangente das estratégias de otimização baseadas em heurísticas, com o intuito de navegar por espaços complexos de busca, e encontrar soluções ótimas ou aproximadas para problemas difíceis.

O Hill Climbing, com uma abordagem direta e baseada em gradientes, é eficiente para encontrar rapidamente máximos locais, mas a sua eficácia é limitada em paisagens complexas onde máximos globais são o objetivo. A simplicidade do HC torna-o numa escolha rápida e fácil para problemas mais simples ou quando uma solução aproximada é aceitável. No entanto, a sua tendência a ficar preso em máximos locais pode ser uma limitação significativa em problemas mais complexos.

Já o Simmulated Annealing expande as capacidades do HC ao introduzir um elemento de aleatoriedade controlado por uma variável, a temperatura, que permite escapar de máximos locais e potencialmente encontrar máximos globais, sendo esta condicionada pela taxa de arrefecimento adequada e tempo de execução suficiente. Essa capacidade de alternar entre exploração e explotação com base na temperatura torna o SA numa ferramenta mais poderosa para problemas onde o espaço de pesquisa do problema é mais desafiador e onde uma solução exata é mais crítica.

Finalmente, os Algoritmos Genéticos, fazem uso de procedimentos inspirados na seleção natural para criar uma população de soluções que evoluem ao longo do tempo. São particularmente eficazes para explorar simultaneamente diferentes regiões do espaço de pesquisa, e mantêm uma diversidade genética que ajuda a evitar a convergência prematura para máximos locais. Com mecanismos de seleção, cruzamento e mutação, os AGs são robustos e flexíveis, capazes de encontrar soluções de alta qualidade em paisagens complexas e multifacetadas.

Ao interligar estes algoritmos, observamos que cada um oferece vantagens únicas que podem ser mais adequadas a certos tipos de problemas:

- O HC é ideal para rapidez e simplicidade;
- O SA para um equilíbrio mais refinado entre exploração e explotação,
- O AG para uma busca robusta e diversificada.

Em última análise, o conhecimento acumulado pela aplicação desses métodos heurísticos de otimização reforça a ideia de que não existe uma solução única para todos os problemas. Em vez disso, um entendimento profundo do problema à mão, juntamente com uma análise cuidadosa das características de cada algoritmo, pode orientar a escolha da estratégia mais adequada para encontrar soluções eficientes e eficazes.

Dada a carga de trabalho e as soluções apresentadas face ao problema em questão, autoavaliamos o nosso trabalho com 17,00 valores.

8. Anexos

8.1 Hill Cimbing (HC)

```
%% Inicialização do ambiente
fx = @(x) 4 * (sin(5 * pi * x + 0.5).^6) .* exp(log2((x -
0.8).^2));
hold off;
x=linspace(0,1,100);
x_array = {}; % Array que guarda a posição, (evolução do x).
j = 1; % Iterados de posições no array de evolução.
re = 10; % Número de reinicializações do algorítmo.
y=fx(x);
plot(x,y,'b');
hold
x_current = rand;
plot(x_current, fx(x_current),'*r');
xlabel('x');
ylabel('F(x)');
title('Algorítmo da Subida da Colina');
t = 1; % Contador de iterações.
t_max = 400; % Número máximo de iterações.
delta = 1/200; % Intervalo de pesquisa, (Vizinhança).

% Algorítmo de pesquisa na vizinhança.

while(t <= t_max)
    x_new = x_current + delta * (2 * rand -1); % Novo ponto gerado
e ponto de comparação com o anterior.
    if(fx(x_current) < fx(x_new))
        x_current = x_new; % Troca a posição do x atual para o
novo x.
        data.fx = fx(x_current);
        data.x = x_current;
        data.time = t;
        x_array{j} = data; % Guarda a posição do x no array de
evolução.
        j = j + 1;
    end
    plot(x_current, fx(x_current),'*r');
    t = t + 1;
end

% Gráfico de evolução da posição do x ao longo do tempo.
figure;
plot(cell2mat(cellfun(@(x) x.time, x_array, 'UniformOutput',
false)), cell2mat(cellfun(@(x) x.x, x_array, 'UniformOutput',
false)), 'g');
xlabel('Tempo');
ylabel('Posição de x');
title('Evolução da posição de x ao longo do tempo');

% Gráfico de evolução imagem de x ao longo do tempo.
```

```
figure;
plot(cell2mat(cellfun(@(x) x.time, x_array, 'UniformOutput',
false)), cell2mat(cellfun(@(x) x.fx, x_array, 'UniformOutput',
false)), 'g');
xlabel('Tempo');
ylabel('F(x)');
title('Evolução da imagem de x ao longo do tempo');
```

8.2 Simulated Annealing (SA)

```
%% Inicialização do ambiente
fx = @(x) 4 * (sin(5 * pi * x + 0.5).^6) .* exp(log2((x -
0.8).^2));
hold off;
x=linspace(0,1,100);
y=fx(x);
plot(x, y, 'b');
hold on
x_current = rand;
plot(x_current, fx(x_current),'*r');
xlabel('x');
ylabel('F(x)');
title('Algorítmo do Simulated Annealing');
t_max = 400; % Número máximo de iterações. % (40 * 10 = 400
iterações no total).
t = 1; % Contador de iterações.
nRep = 10; % Número de repetições para cada valor de
temperatura.
delta = 1/40; % Intervalo de pesquisa, (Vizinhança).
T = 90; % Temperatura inicial. (Máximo).
alfa = 0.94; % Valor de decaímento da temperatura.
values = {}; % Array que guarda os valores necessários para
desenhar os gráficos.
i = 1; % Contador do array.

% Algorítmo de pesquisa na vizinhança.
while(t <= t_max)
    rep = 1; % Contador de repetições.
    while(rep <= nRep)
        x_new = x_current + delta * (2 * rand -1); % Novo ponto
gerado e ponto de comparação com o anterior.
        % Garante que o novo ponto gerado se encontra dentro do
domínio.
        if(x_new >= 0 && x_new <= 1)
            dE = fx(x_new) - fx(x_current); % Gradiente de
energia.
            p = 1/(1+exp(abs(dE)/T)); % Probabilidade de aceitar
um valor pior.
            % Maximização.
            if(dE > 0)
                x_current = x_new;
                imgx = fx(x_new);
            % Caso em que aceita um valor pior, (Minimização).
            elseif(rand < p)
```

```
                x_current = x_new;
                imgx = fx(x_new);
            end
            % Guarda os valores numa estrutura para desenhar os
gráfico.
            data.x = x_current;
            data.y = fx(x_current);
            data.de = dE;
            data.t = T;
            data.p = p;
            values{i} = data;
            i = i + 1;
            rep = rep + 1;
        end
    end
    T = T * alfa; % Diminuição da temperatura.
    t = t + 1;
end

% Desenha os gráficos.

% Gráfico do algorítmo.
for j = 1:(nRep * t_max)
    plot(values{j}.x, values{j}.y,'*r');
end

figure;
t = tiledlayout(5, 1, 'TileSpacing', 'compact');

% Gráfico da evolução de x.
ax1 = nexttile;
for j = 1:(nRep * t_max)
    plot(ax1, j, values{j}.x, '*r');
    hold(ax1, 'on');
end
xlabel(ax1, 'Iteração');
ylabel(ax1, 'x');
title(ax1, 'Evolução de x');

% Gráfico da evolução de y.
ax2 = nexttile;
for j = 1:(nRep * t_max)
    plot(ax2, j, values{j}.y, '*b');
    hold(ax2, 'on');
end
xlabel(ax2, 'Iteração');
ylabel(ax2, 'F(x)');
title(ax2, 'Evolução de y');

% Gráfico da evolução de dE.
ax3 = nexttile;
for j = 1:(nRep * t_max)
    plot(ax3, j, values{j}.de, '*g');
    hold(ax3, 'on');
end
xlabel(ax3, 'Iteração');
```

```
ylabel(ax3, 'dE');
title(ax3, 'Evolução da Energia');

% Gráfico da evolução de dE.
ax4 = nexttile;
for j = 1:(nRep * t_max)
    plot(ax4, j, values{j}.t, '*y');
    hold(ax4, 'on');
end
xlabel(ax4, 'Iteração');
ylabel(ax4, 'T');
title(ax4, 'Evolução da Temperatura');


% Gráfico da evolução da Probabilida.
ax5 = nexttile;
for j = 1:(nRep * t_max)
    plot(ax5, j, values{j}.p, '*r');
    hold(ax5, 'on');
end
xlabel(ax5, 'Iteração');
ylabel(ax5, 'P');
title(ax5, 'Evolução da Probabilidade');
```

8.2 Algoritmo Genético (AG)

```
% Plot da Função e valor máximo
limits=[0,1]; % Limites da pesquisa
set(0,'defaultlinelinewidth',2);
ezplot('4*(sin(5*pi*x+0.5)^6)*exp(log2((x-0.8)^2))',limits)
title('Algorítmo Genético');
axis([0,1,-0.1,2]); % Delimitar os eixos
hold on
% Plots solução do máximo
plot(0.066,1.6332,'sk','Linewidth',2,'markersize',6,'markerfacec
olor','r');
% Definições de variáveis
pop_size=5; % Tamanho da populaçao
lchrome=12; % Tamanho do cromossoma
maxgen=80; % Número maximo de gerações
p_cross=0.8; % Probabilidade de cruzamento
p_mutation=0.04; % Probabilidade de mutação
gen=1; % Contador das gerações

% Inicializa a população e guarda os respetivos crossomas na
matriz CHROME.
CHROME = init_pop(pop_size, lchrome);

% Calcular aptidão para cada indivíduo na população.
for i = 1:pop_size
    fitnessValues = bin2dec(num2str(CHROME(i, :))) / (2^lchrome
- 1);
end
```

```
% O vetor 'POP' contém as aptidões da população.
POP = fitnessValues;

% Calcula a aptidão e o valor da variável de decisão.
[CHROME, POP, POP_x] = dec_pop(pop_size, lchrome, limits);

% Calcula as métricas estatísticas da população inicial.
[sumfit, best_idx, max_value, average] = statist(POP);

% Identifica o melhor indivíduo inicial e qual a sua aptidão.
best_chrome = CHROME(best_idx, :);
best_fitness = max_value;

% Inicializa vetores para armazenar os dados de progresso.
best_fitness_over_time = zeros(maxgen, 1);
average_fitness_over_time = zeros(maxgen, 1);

% Plot da função objetivo como pano de fundo.
figure(1);
fplot(@(x) objective_function(x), limits, 'b-', 'LineWidth', 1);
title('Evolução da População no Espaço de Soluções');
xlabel('x');
ylabel('Aptidão');
hold on;

% Ciclo principal do algoritmo genético.
for gen = 1:maxgen
    % Gera a nova população.
    CHROME = generate(pop_size, POP, CHROME, lchrome, p_cross,
p_mutation);

    % Avalia a nova população.
    [POP, POP_x] = evaluate_population(CHROME, lchrome, limits);

    % Calcula estatísticas da nova população.
    [sumfit, best_idx, max_value, average] = statist(POP);

    % Plota a função objetivo novamente como pano de fundo.
    cla; % Limpa o gráfico atual.
    fplot(@(x) objective_function(x), limits, 'b-', 'LineWidth',
1);
    hold on;

    % Plota as soluções da população.
    scatter(POP_x, POP, 'g*');
    plot(POP_x(best_idx), max_value, 'ro', 'MarkerFaceColor',
'r'); % Destaca a melhor solução.

    % Armazena dados para gráficos de progresso.
    best_fitness_over_time(gen) = max_value;
    average_fitness_over_time(gen) = average;

    % Atualiza o título com o número da geração.
    title(sprintf('Evolução da População no Espaço de Soluções -
Geração %d', gen));
```

```
    % Pausa breve para a visualização da geração atual.
    pause(0.1);

    hold off;
end

% Plot final dos gráficos de progresso.
figure;
subplot(2, 1, 1);
plot(best_fitness_over_time, 'r-', 'LineWidth', 2);
title('Melhor Aptidão por Geração');
xlabel('Geração');
ylabel('Aptidão');

subplot(2, 1, 2);
plot(average_fitness_over_time, 'b-', 'LineWidth', 2);
title('Aptidão Média por Geração');
xlabel('Geração');
ylabel('Aptidão');

%% Funções necessárias para a execução do algoritmo.

% Função que inicializa a populaçao
function CHROME = init_pop(pop_size, lchrome)
    % CHROME - Matriz da população atual
    CHROME = randi([0, 1], pop_size, lchrome);
end

% Função que calcula a aptidão e o valor da variável de decisão.
function [CHROME, POP, POP_x] = dec_pop(pop_size, lchrome,
limits)
    % Inicialização da população com valores aleatórios dentro
dos limites.
    CHROME = round(rand(pop_size, lchrome));

    % Inicializa o vetor de aptidão e o vetor de variáveis de
decisão.
    POP = zeros(1, pop_size);
    POP_x = zeros(1, pop_size);

    % Calcula a aptidão para cada cromossoma.
    for i = 1:pop_size
        % Converte o cromossoma binário num valor real dentro
dos limites.
        POP_x(i) = bin2dec(num2str(CHROME(i, :))) / (2^lchrome -
1) * (limits(2) - limits(1)) + limits(1);
        POP(i) = objective_function(POP_x(i));
    end
end

% Função objetivo.
function y = objective_function(x)
    y = 4 * (sin(5 * pi * x + 0.5).^6) .* exp(log2((x -
0.8).^2));
end
```

```
% Função que estatísticas da nova população.
function [sumfit, best, max_value, average] = statist(POP)
    sumfit = sum(POP); % Soma das aptidões.
    [max_value, idx_best] = max(POP); % Melhor aptidão e índice.
    best = idx_best; % Índice do melhor cromossoma.
    average = mean(POP); % Média das aptidões.
end

% Seleciona os indivíduos pais da próxima geração com base na
aptidão de cada indivíduo.
function index = roulette_selection(POP)
    cumulative_sum = cumsum(POP);
    roulette_pick = rand * cumulative_sum(end);
    index = find(cumulative_sum >= roulette_pick, 1, 'first');
end

% Realiza um cruzamento de um ponto entre dois pais para criar
dois filhos.
% O ponto de cruzamento é escolhido aleatoriamente.
function [child1, child2] = single_point_crossover(parent1,
parent2, p_cross)
    if rand <= p_cross
        % Verifica se o cruzamento ocorrerá, com base na
probabilidade de cruzamento.
        point = randi(length(parent1) - 1);
        child1 = [parent1(1:point), parent2(point+1:end)];
        child2 = [parent2(1:point), parent1(point+1:end)];
    else
        % Se não houver cruzamento, os filhos são cópias dos
pais.
        child1 = parent1;
        child2 = parent2;
    end
end

% Aplica mutação a um indivíduo (cromossoma).
function individual = mutate(individual, p_mutation)
    for i = 1:length(individual)
        % Verifica se a mutação ocorrerá para cada gene.
        if rand <= p_mutation
            individual(i) = 1 - individual(i);
        end
    end
end

% Gera uma nova população de cromossomas (CHROME) usando os
métodos de seleção, cruzamento e mutação.
function CHROME = generate(pop_size, POP, CHROME, lchrome,
p_cross, p_mutation)
    new_CHROME = zeros(size(CHROME));
    for i = 1:2:pop_size
        % Seleção.
        index1 = roulette_selection(POP);
        index2 = roulette_selection(POP);

        % Cruzamento.
```

```
        [child1, child2] = single_point_crossover(CHROME(index1,
:), CHROME(index2, :), p_cross);

        % Mutação.
        child1 = mutate(child1, p_mutation);
        child2 = mutate(child2, p_mutation);

        new_CHROME(i, :) = child1;
        if i+1 <= pop_size
            new_CHROME(i+1, :) = child2;
        end
    end
    CHROME = new_CHROME;
end

% Calcula a aptidão da população.
function [POP, POP_x] = evaluate_population(CHROME, lchrome,
limits)
    pop_size = size(CHROME, 1); % Número de indivíduos na
população.
    POP = zeros(1, pop_size); % Vetor de aptidão.
    POP_x = zeros(1, pop_size); % Vetor de valores da variável
de decisão.

    % Itera sobre a população para calcular a aptidão de cada
cromossoma.
    for i = 1:pop_size
        % Converte o cromossoma binário em valor real dentro dos
limites.
        x = bin2dec(num2str(CHROME(i, :))) / (2^lchrome - 1);
        x = x * (limits(2) - limits(1)) + limits(1);
        POP_x(i) = x; % Armazena o valor real da variável de
decisão.
        POP(i) = objective_function(x); % Calcula e armazena a
aptidão.
    end
end
```