INSTITUTO TECNOLÓGICO DE AERONÁUTICA
CT-213

Laboratório 3: Otimização com Métodos de Busca Local

Professor:
Marcos Ricardo Omena de Albuquerque Máximo

Aluna:
Thayná Pires Baldão

São José dos Campos, 3 de abril de 2021

1 *Gradient Descent*

Para implementar o algorimo *Gradient Descent*, seguiu-se o pseudo-código apresentado no Slide 41 da Aula 3 de CT-213. As condições de parada verificadas a cada iteração do algoritmo foram:

- Se o custo do `theta` atual era superior ou igual a um `epsilon`; e
- Se o número de iterações do algoritmo não tinha excedido `max_iterations`.

Para fazer esta última verificação foi preciso criar uma variável auxiliar denominada `iteration`, que era atualizada a cada iteração do código.

Além disso, a função `gradient_descent` do laboratório requisitava que se retornasse um *array* com todos os pontos visitados durante a execução do algoritmo. Por causa disso, a cada iteração do algoritmo, adicionou-se o `theta` visitado a um *array* denominado `history` que, posteriormente, foi retornado pela função.

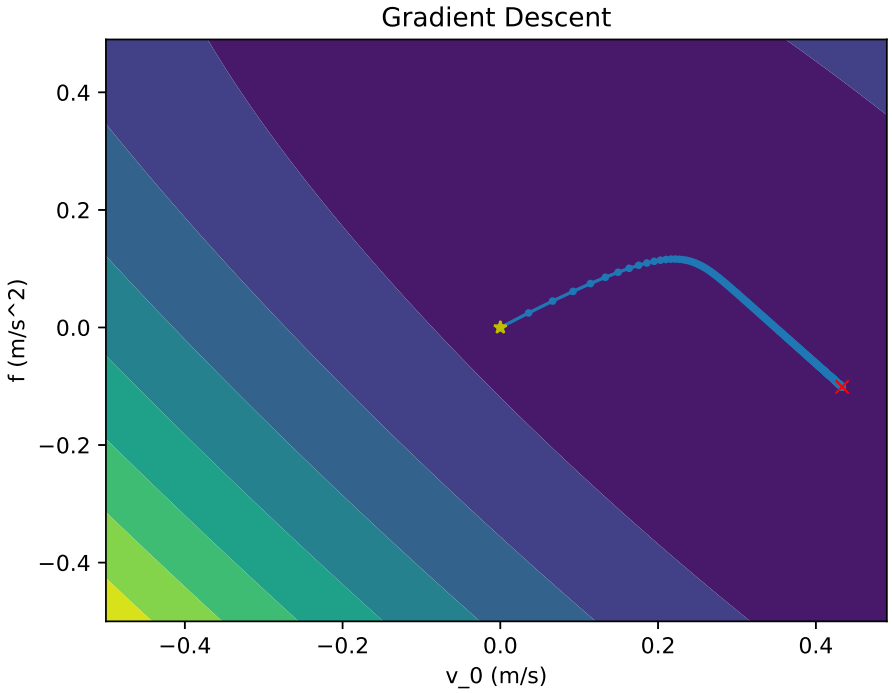

Figura 1: Trajetória de otimização usando *Gradient Descent*.

2 *Hill Climbing*

Para implementar o algorimo *Hill Climbing*, seguiu-se o pseudo-código apresentado no Slide 46 da Aula 3 de CT-213, com algumas modificações. Isso porque, no pseudo-código apresentado na aula, o *Hill Climbing* estava tentando resolver um problema de otimização envolvendo maximização e não minimização. Todavia, sabe-se que esses problemas são simétricos, portanto, para que o *Hill Climbing* funcionasse com um problema de minimização, bastou inverter as desigualdades checadas pelo algoritmo, bem como inicializar o custo com $+\infty$ ao invés de $-\infty$.

Além disso, criou-se as variáveis auxiliares `theta_J`, `neighbor_J` e `best_J` para armazenar, respectivamente, o custo do `theta`, do `neighbor` e do `best`, com o intuito de evitar recalcular esses custos múltiplas vezes e, desta forma, melhorar a performance do algoritmo.

Para calcular o vetor de vizinhos do algoritmo seguiu-se a recomendação do roteiro de pegar 8 vizinhos igualmente espaçados em ângulo e a uma distância `delta` do ponto atual. Para implementar essa função,

simplesmente criou-se um laço que, a cada iteração, obtinha um ponto vizinho utilizando o ângulo da iteração (`angle`), `delta` e geometria básica. Então, o ponto obtido era adicionado ao *array* `neighbors` e o `angle` era incrementado em 45°. O laço persistia até que `angle` varesse todo o círculo trigonométrico.

Ademais, as condições de parada verificadas a cada iteração do algoritmo foram:

- Se o custo do `theta` atual era superior ou igual a um `epsilon`; e
- Se o número de iterações do algoritmo não tinha excedido `max_iterations`.

Para fazer esta última verificação foi preciso criar uma variável auxiliar denominada `iteration`, que era atualizada a cada iteração do código.

Por fim, a função `hill_climbing` do laboratório requisitava que se retornasse um *array* com todos os pontos visitados durante a execução do algoritmo. Por causa disso, a cada iteração do algoritmo, adicionou-se o `theta` visitado a um *array* denominado `history` que, posteriormente, foi retornado pela função.

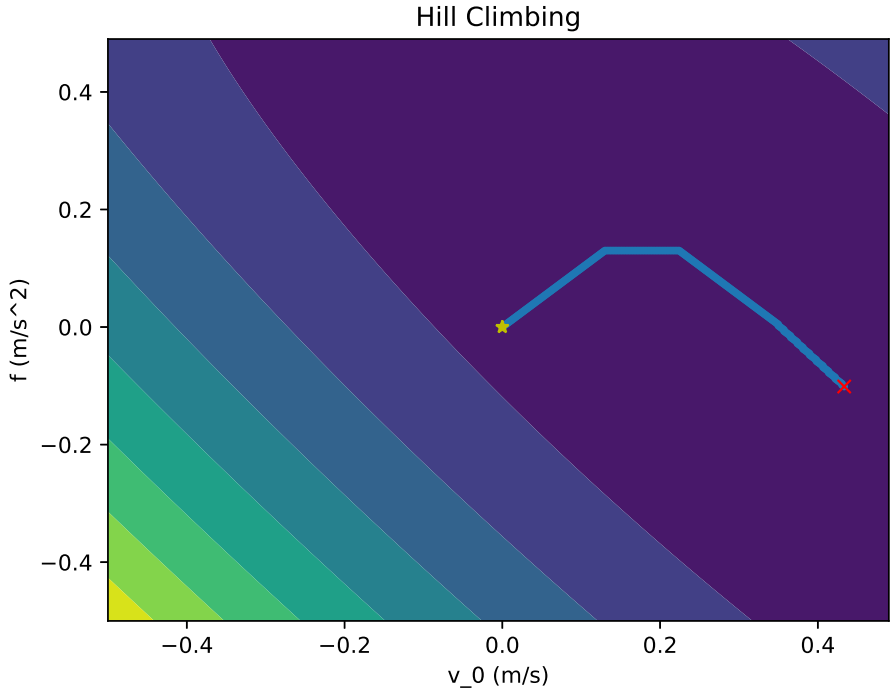

Figura 2: Trajetória de otimização usando *Hill Climbing*.

3 *Simulated Annealing*

Para implementar o algorimo *Simulated Annealing*, seguiu-se o pseudo-código apresentado no Slide 51 da Aula 3 de CT-213, com algumas modificações. Isso porque, no pseudo-código apresentado na aula, o *Simulated Annealing* estava tentando resolver um problema de otimização envolvendo maximização e não minimização. Todavia, sabe-se que esses problemas são simétricos, portanto, para que o *Simulated Annealing* funcionasse com um problema de minimização, bastou inverter o sinal do `deltaE` calculado a partir do custo do `theta` e do `neighbor`.

Para calcular o vizinho aleatório do algoritmo seguiu-se a recomendação do roteiro de pegar um ponto a uma distância `delta` do ponto atual e com ângulo amostrado aleatoriamente com distribuição uniforme no intervalo $(-\pi, \pi)$. Para calcular o escalonamento de temperatura também seguiu-se a recomendação do roteiro de utilizar a seguinte equação: `temperature` = `temperature0/(1 + beta*(iteration**2))`.

Ademais, as condições de parada verificadas a cada iteração do algoritmo foram:

- Se o custo do `theta` atual era superior ou igual a um `epsilon`; e
- Se o número de iterações do algoritmo não tinha excedido `max_iterations`.

Para fazer esta última verificação foi preciso criar uma variável auxiliar denominada `iteration`, que era atualizada a cada iteração do código.

Por fim, a função `simulated_annealing` do laboratório requisitava que se retornasse um *array* com todos os pontos visitados durante a execução do algoritmo. Por causa disso, a cada iteração do algoritmo, adicionou-se o `theta` visitado a um *array* denominado `history` que, posteriormente, foi retornado pela função.

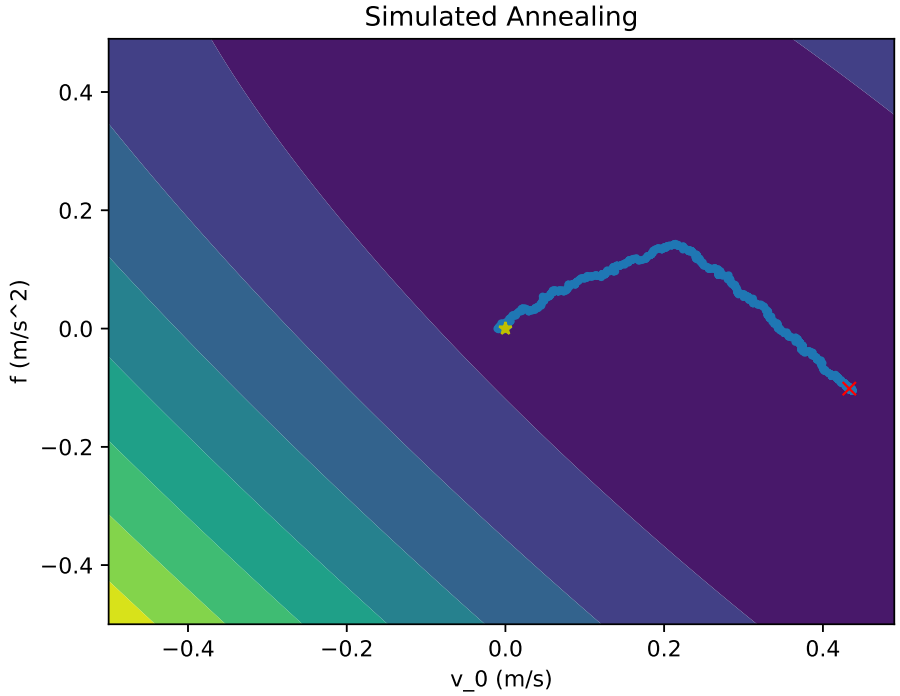

Figura 3: Trajetória de otimização usando *Simulated Annealing*.

4 Comparação entre os algoritmos de otimização com métodos de busca local

Tabela 1: Solução encontrada por cada um dos algoritmos de otimização com métodos de busca local.

Algoritmo	**Solução** $[v_0\ (m/s),\ f\ (m/s^2)]$
Least Squares	[0.43337277 −0.10102096]
Gradient Descent	[0.43337067 −0.10101846]
Hill Climbing	[0.43341125 −0.10119596]
Simulated Annealing	[0.43327855 −0.10154538]

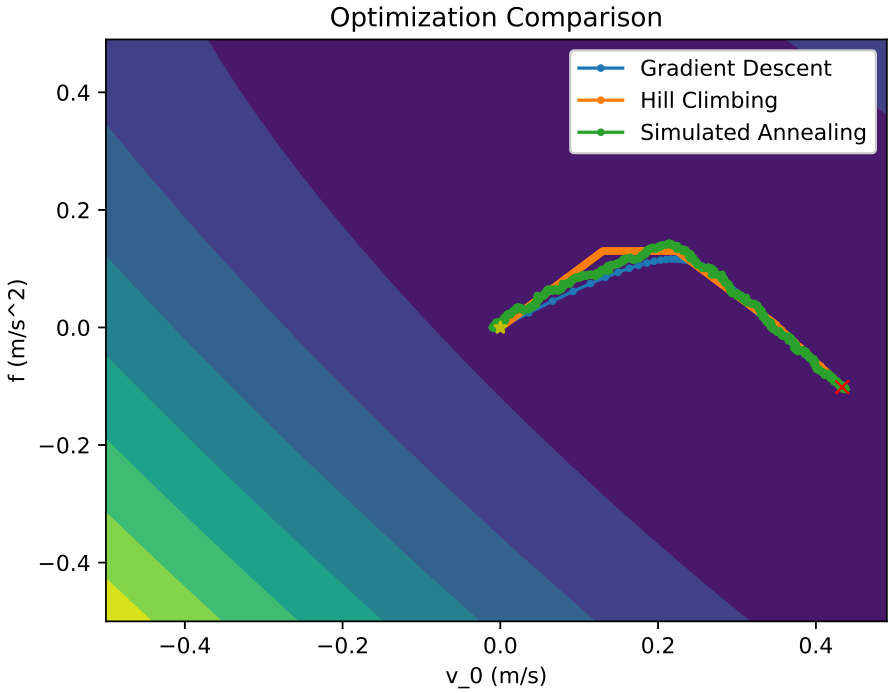

Figura 4: Comparação de trajetórias de otimização usando *Gradient Descent*, *Hill Climbing* e *Simulated Annealing*.

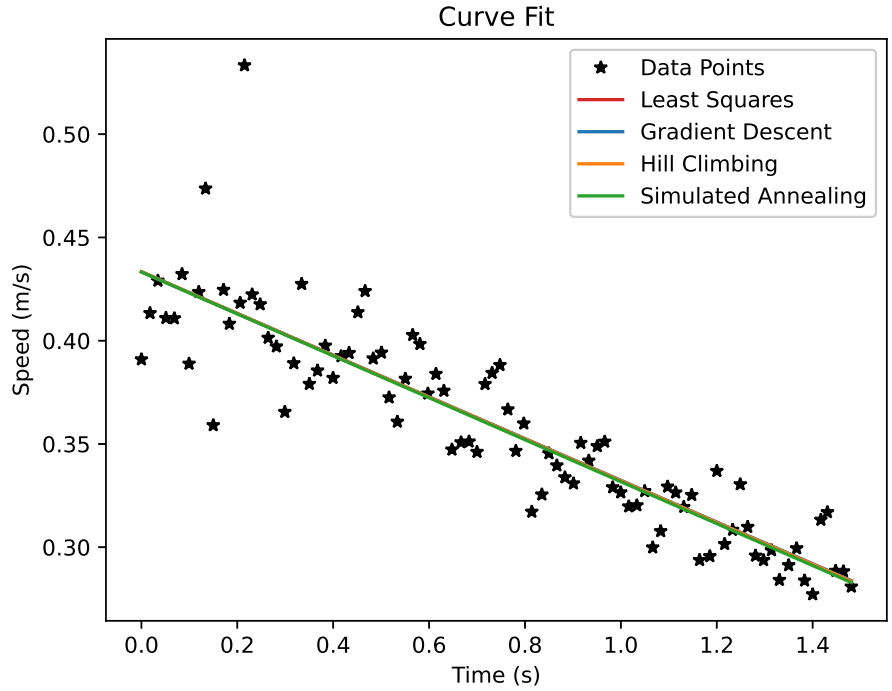

Figura 5: Comparação das curvas de fit usando *Gradient Descent*, *Hill Climbing* e *Simulated Annealing*.