Instituto Tecnológico de Aeronáutica – ITA
Inteligência Artificial para Robótica Móvel – CT-213
Professor: Marcos Ricardo Omena de Albuquerque Maximo

Laboratório 3 – Otimização com Métodos de Busca Local

1. Introdução

Nesse laboratório, seu objetivo é implementar algoritmos de otimização baseados em busca local, a saber Descida do Gradiente, *Hill Climbing* e *Simulated Annealing*. Os métodos serão testados em um problema em que se usa regressão linear para obter parâmetros físicos relativos ao movimento de uma bola. No caso, trata-se apenas de um problema brinquedo, dado que esse problema especificamente tem solução analítica, logo o Método dos Mínimos Quadrados (MMQ) o resolve mais facilmente.

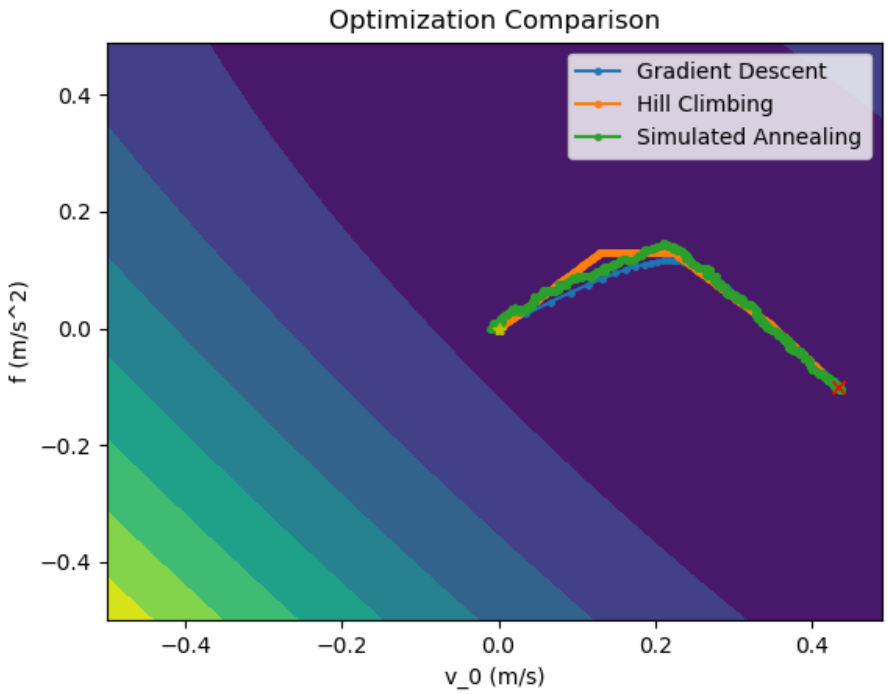

Figura 1: comparação de trajetórias de otimização usando Gradient Descent, Hill Climbing e Simulated Annealing.

2. Descrição do Problema

O problema ser resolvido é a otimização de funções matemáticas (em que é possível obter amostras da função e de sua derivada) usando os métodos descritos, assim sua implementação **não** deve ser específica para o caso de teste. As descrições dos algoritmos que serão implementados podem ser vistas nos slides do curso.

No caso de teste, o problema específico a ser resolvido é a determinação do coeficiente de desaceleração de uma bola em movimento num campo de futebol de robôs. A bola em movimento perde energia devido a um fenômeno conhecido como *rolling friction*. Conforme explicado em aula, pode-se determinar o coeficiente de desaceleração através do seguinte algoritmo:

1. Usar câmera e visão computacional para obter posições (x, y) da bola em cada instante.
2. Calcular velocidades em x e y usando diferenças finitas centradas (exceto no primeiro e último elementos, em que deve-se usar derivadas forward e backward, respectivamente):

$$v_x[k] = (x[k+1] - x[k-1])/(t[k+1] - t[k-1])$$

$$v_y[k] = (y[k+1] - y[k-1])/(t[k+1] - t[k-1])$$

3. Calcular $v[k] = \sqrt{v_x^2[k] + v_y^2[k]}$.
4. Obter v_0 e f através de uma otimização com função de custo:

$$J([v_0, f] = \sum_{k=1}^{n} (v_0 + ft[k] - v[k])^2$$

Desse modo, tem-se os seguintes parâmetros a serem otimizados:

$\theta_0 = v_0$ e $\theta_1 = f \Rightarrow \theta = [v_0 \ f]^T$

3. Código Base

Juntamente com o código base, foi entregue o arquivo data.txt, que contém dados reais do movimento de uma bola no campo de futebol de robôs do Very Small Size (VSS). Esses dados foram obtidos pelo aluno Thiago Filipe de Medeiros da COMP-19 durante uma competição. Para isso, usou-se o setup do VSS, incluindo os algoritmos de visão computacional da ITAndroids.

O código base entregue já carrega esses dados automaticamente e os pré-processa de acordo com o procedimento explicado na seção anterior. Além disso, resolve a otimização com uso do MMQ a fim de fornecer valores esperados para v_0 e f. O código base também já executa os algoritmos que você vai implementar e apresenta gráficos para que você possa verificar sua implementação.

O arquivo “main” desse laboratório é o `ball_fit.py`.

4. Tarefas

Você deve implementar os algoritmos Descida do Gradiente, *Hill Climbing* e *Simulated Annealing*. Para isso, implemente as seguintes funções:

- `gradient_descent()` em `gradient_descent.py`.
- `hill_climbing()` em `hill_climbing.py`.
- `simulated_annealing()` em `simulated_annealing.py`.

As implementações desses algoritmos de otimização devem ser as mais genéricas possível e não usar informações específicas do problema de *fit* da dinâmica da bola. Perceba que as condições de parada de todos os algoritmos envolvem um limiar para o custo (i.e. o algoritmo pára quando $J(\theta) < \varepsilon$) e um número máximo de iterações.

Os métodos de otimização requerem algumas funções auxiliares dependentes do problema, que estão em `ball_fit.py`, a saber:

- Descida do Gradiente: `cost_function()` e `gradient_function()`.
- Hill Climbing: `cost_function()` e `neighbors()`.
- Simulated Annealing: `cost_function()`, `random_neighbor()` e `schedule()`.

As implementações de `cost_function()` e `gradient_function()` foram fornecidas como exemplo, mas as demais devem ser implementadas pelo aluno. Para os vizinhos de um ponto no *Hill Climbing*, use uma estratégia 8-conectada: pegue 8 vizinhos igualmente espaçados em ângulo e a uma distância Δ do ponto atual, conforme mostra a Figura 2. Para o vizinho aleatório do Simulated Annealing, considere um ponto a uma distância Δ do ponto atual e com ângulo amostrado aleatoriamente com distribuição uniforme no intervalo $(-\pi, \pi)$. Para o escalonamento de temperatura, use a seguinte equação:

$$T = T_0/(1 + \beta i^2)$$

em que T_0 e β são hiperparâmetros e i é a iteração atual (começando em 0). Observe que valores adequados de Δ, T_0 e β já estão presentes em `ball_fit.py`.

Para comprovar o correto funcionamento da sua implementação, basta incluir no relatório as figuras geradas pelo `ball_fit.py` e uma tabela com as soluções encontradas por cada algoritmo.

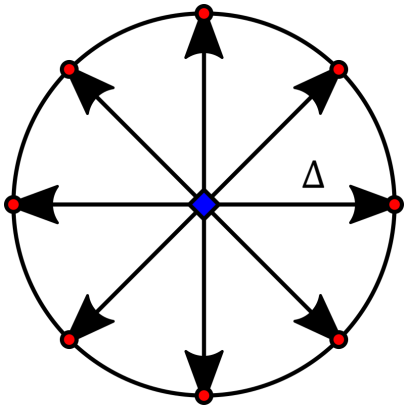

Figura 2: vizinhos em estratégia 8-conectada usada no *Hill Climbing*.

5. Entrega

A entrega consiste do código e de um relatório, submetida através do Google Classroom. Modificações nos arquivos do código base são permitidas, desde que o nome e a interface dos scripts “main” não sejam alterados. A princípio, não há limitação de número de páginas para o relatório, mas pede-se que seja sucinto. O relatório deve conter:

- Breve descrição em alto nível da sua implementação.
- Figuras que comprovem o funcionamento do seu código.

Por limitações do Google Classroom (e por motivo de facilitar a automatização da correção), entregue seu laboratório com todos os arquivos num único arquivo **.zip** (**não** utilize outras tecnologias de compactação de arquivos) com o seguinte padrão de nome: "**<login_email_google_education>_labX.zip**". Por exemplo, no meu caso, meu login Google Education é **marcos.maximo**, logo eu entregaria o lab 3 como "**marcos.maximo_lab3.zip**". **Não** crie subpastas para os arquivos da sua entrega, **deixe todos os arquivos na "raiz" do .zip**. Os relatórios devem ser entregues em formato **.pdf**.

6. Dicas

- Para criar um vetor com dois elementos em NumPy, faça:

```
vector = np.array([1, 2])
```

- Para amostrar um valor aleatoriamente com distribuição uniforme no intervalo (a, b), use:

```
X = random.uniform(a, b)
```