Laboratório 3
Otimização com Métodos de Busca Local

CT-213
Prof. Marcos Ricardo Omena de Albuquerque Maximo

Carlos Matheus Barros da Silva
carlosmatheusbs@gmail.com

Março - 2019

Introdução

Foram estudados e implementados três métodos de otimização que possuem algoritmos baseados em busca local. Esses métodos foram Gradient Descent, Hill Climbing e Simulated Anneling.

Os métodos foram testados em um problema proposto no roteiro dessa atividade. Esse problema utilizou regressão linear para ober parâmetros físicos relativos ao movimento de uma bola.

Como o problema tratado possui solução analítica, o experimento é a titulo educacional, já que, dessa forma, pode-se obter facilmente sua solução pelo Método dos Mínimos Quadrados (MMQ).

De maneira geral, o Laboratório foi desenvolvido com sucesso. Em cada uma das sessões seguintes encontram-se o respectivo desenvolvimento de um dos métodos estudado.

Gradient Descent

Explicação 1. O código para a o algoritmo do Gradient Descent pode ser visto no Código 1. Para a implementação desse código foram utilizados, também as funções no Código 2 e no Código 3.

O resultado desse algoritmo pode ser visto no hitórico de *theta* represetado pelo Código 4 e pode ser visto também na análise da trajetória apresentada na Figura 1.

```
    algorithm = "gradient_descent"

    theta = theta0
    history = [theta0]
    i = 0
    while not check_stopping_condition(
    max_iterations, i, cost_function,
    epsilon, theta):
        theta = theta - alpha *
    gradient_function(theta)
        history.append(theta)
        i += 1

    create_history_file(history, algorithm)

    return theta, history
```

Código 1: Interior da função *Gradient Descent*.

```
def create_history_file(history, algorithm):
    f = open(algorithm + ".txt", "w+")
    i = 1
    for theta in history:
        f.write("i = " + str(i) + " => " +
    str(theta) + "\n")
        i += 1
```

Código 2: Código para criação do arquivo com o hitórico de *theta*.

```
def check_stopping_condition(max_iterations,
     i, cost_function, epsilon, theta):
    return i > max_iterations or
    cost_function(theta) < epsilon
```

Código 3: Código para checagem da condição de parada.

```
i = 1 => [0. 0.]
i = 2 => [0.03584969 0.02468302]
i = 3 => [0.06628493 0.04489148]
i = 4 => [0.09217881 0.06135615]
i = 5 => [0.11426295 0.07468929]
i = 6 => [0.13315044 0.08540385]
i = 7 => [0.14935503 0.09392958]
i = 8 => [0.16330723 0.10062648]
i = 9 => [0.17536785 0.10579616]
i = 10 => [0.18583924 0.10969127]
i = 993 => [ 0.43337047 -0.10101822]
i = 994 => [ 0.43337049 -0.10101825]
i = 995 => [ 0.43337052 -0.10101828]
i = 996 => [ 0.43337055 -0.10101831]
i = 997 => [ 0.43337057 -0.10101834]
i = 998 => [ 0.4333706  -0.10101837]
i = 999 => [ 0.43337062 -0.1010184 ]
i = 1000 => [ 0.43337065 -0.10101843]
i = 1001 => [ 0.43337067 -0.10101846]
i = 1002 => [ 0.4333707  -0.10101849]
```

Código 4: Histórico de *theta* para o algoritmo *Gradient Descent* para as 10 primeiras iterações e para as 10 útilmas

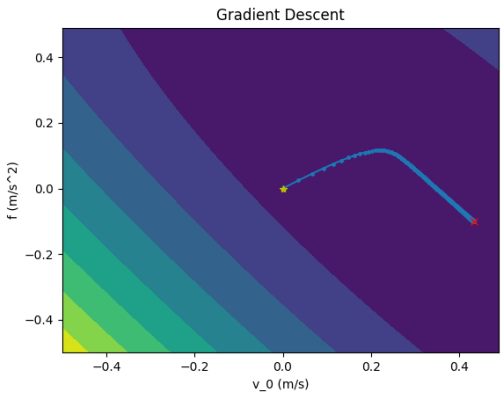

Figura 1: Trajetório com o método *Gradient Descent*.

Hill Climbing

Explicação 2. O código para a o algoritmo do Hill Climbing pode ser visto no Código 5. Para a implementação desse código foram utilizados, também as funções no Código 2 e no Código 3.

O resultado desse algoritmo pode ser visto no hitórico de *theta* represetado pelo Código 6 e pode ser visto também na análise da trajetória apresentada na Figura 2.

```
    algorithm = "hill_climbing"

    theta = theta0
    history = [theta0]
```

```
5       i = 0
6       while not check_stopping_condition(
        max_iterations, i, cost_function,
        epsilon, theta):
7           best = None  # J(None) = -inf
8           for neighbor in neighbors(theta):
9               # print(cost_function(neighbor))
10              # print(cost_function(best))
11              if cost_function(neighbor) <
        cost_function(best):
12                  best = neighbor
13          if cost_function(best) >
        cost_function(theta):
14              history.append(theta)
15              # print(history)
16              break
17          theta = best
18          history.append(theta)
19          # print(history)
20          i += 1
21
22      create_history_file(history, algorithm)
23
24      return theta, history
```

Código 5: Interior da função *Hill Climbing*.

```
i = 1 => [0. 0.]
i = 2 => [0.00141421 0.00141421]
i = 3 => [0.00282843 0.00282843]
i = 4 => [0.00424264 0.00424264]
i = 5 => [0.00565685 0.00565685]
i = 6 => [0.00707107 0.00707107]
i = 7 => [0.00848528 0.00848528]
i = 8 => [0.00989949 0.00989949]
i = 9 => [0.01131371 0.01131371]
i = 10 => [0.01272792 0.01272792]
i = 291 => [ 0.42351176 -0.08929646]
i = 292 => [ 0.42492597 -0.09071068]
i = 293 => [ 0.42634019 -0.09212489]
i = 294 => [ 0.4277544  -0.09353911]
i = 295 => [ 0.42916861 -0.09495332]
i = 296 => [ 0.43058283 -0.09636753]
i = 297 => [ 0.43058283 -0.09836753]
i = 298 => [ 0.43199704 -0.09978175]
i = 299 => [ 0.43341125 -0.10119596]
i = 300 => [ 0.43341125 -0.10119596]
```

Código 6: Histórico de *theta* para o algoritmo *Hill Climbing* para as 10 primeiras iterações e para as 10 útilmas

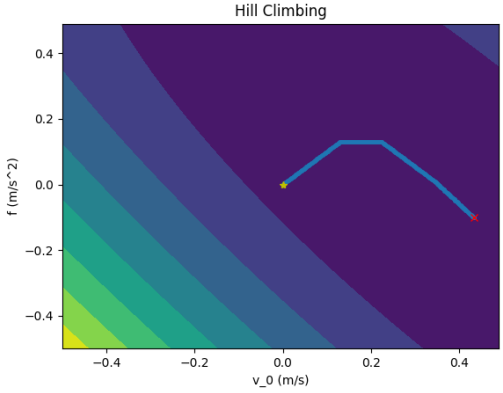

Figura 2: Trajetório com o método *Hill Climbing*.

Simulated Anneling

Explicação 3. O código para a o algoritmo do Simulated Anneling pode ser visto no Código 7. Para a implementação desse código foram utilizados, também as funções no Código 2 e no Código 3.

O resultado desse algoritmo pode ser visto no hitórico de *theta* represetado pelo Código 8 e pode ser visto também na análise da trajetória apresentada na Figura 3.

```
1       algorithm = "simulated_annealing"
2
3       while not check_stopping_condition(
        max_iterations, i, cost_function,
        epsilon, theta):
4           T = schedule(i)
5           # print(T)
6           if T < 0.0:
7               return theta, history
8           neighbor = random_neighbor(theta)
9           deltaE = cost_function(neighbor) -
        cost_function(theta)
10          if deltaE < 0:
11              theta = neighbor
12          else:
13              r = random.uniform(0.0, 1.0)
14              if r >= exp(-deltaE / T):
15                  # print(i, "yes")
16                  theta = neighbor
17              # else:
18                  # print(i, "no")
19
20          history.append(theta)
21          i += 1
22
23      # print(history)
24      create_history_file(history, algorithm)
25
26      return theta, history
```

Código 7: Interior da função *Simulated Anneling*.

```
i = 1 => [0. 0.]
i = 2 => [0. 0.]
i = 3 => [-0.00026044  0.00198297]
i = 4 => [0.00029135 0.00390535]
i = 5 => [0.00051758 0.00589251]
i = 6 => [0.00234559 0.00508112]
i = 7 => [0.00172731 0.00698315]
i = 8 => [0.00368473 0.00739366]
i = 9 => [0.00368473 0.00739366]
i = 10 => [0.00368473 0.00739366]
i = 4991 => [ 0.43294321 -0.10068177]
i = 4992 => [ 0.43294321 -0.10068177]
i = 4993 => [ 0.43294321 -0.10068177]
i = 4994 => [ 0.43294321 -0.10068177]
i = 4995 => [ 0.43294321 -0.10068177]
i = 4996 => [ 0.43294321 -0.10068177]
i = 4997 => [ 0.43294321 -0.10068177]
i = 4998 => [ 0.43294321 -0.10068177]
i = 4999 => [ 0.43294321 -0.10068177]
i = 5000 => [ 0.43294321 -0.10068177]
```

Código 8: Histórico de *theta* para o algoritmo *Simulated Anneling* para as 10 primeiras iterações e para as 10 útilmas

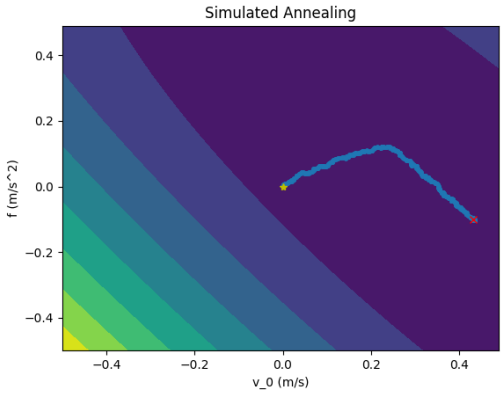

Figura 3: Trajetório com o método *Simulated Anneling*.

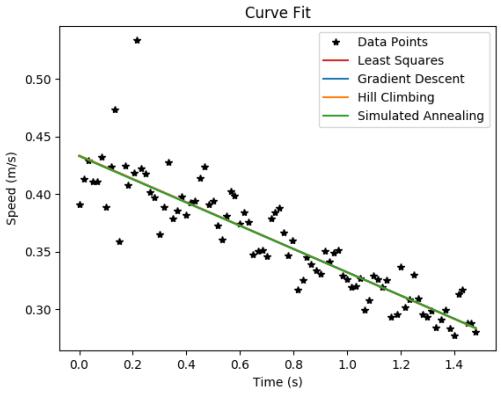

Figura 4: Comparação das cursvas sobre os pontos amostrados.

Comparações

De maneira visual a comparação dos métodos pode ser vista na Figura 4 e na Figura 5.

Como foi observado o método *Hill Climbing* converge mais rapidamente para um mínimo local, enquanto que o método *Simulated Anneling* converge mais devagar, mas apreseta maior capacidade de encontrar um minimo global ao invez de local. Isso se deve ao fato de que randomicamente ele erra a fim de que possa encontrar outros caminhos.

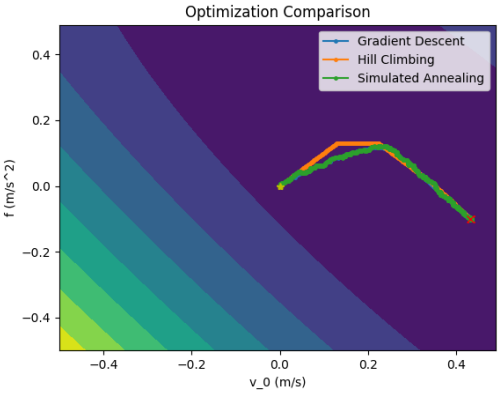

Figura 5: Comparação das trajetórias entre os métodos