1 Introdução

Esse relatório apresenta resultados da implementação dos algorítimos de otimização (descida do gradiente, *hill climbing* e *simulated anealling*) e de métodos auxiliares num código disponibilizado via *Google Classroom*.

2 Métodos Auxiliares

```
def schedule(i):
    retorne temperature0/(1+beta*i**2)

def random_neighbor(theta):
    alpha=random_uniform(-pi,pi)
    retorne [theta[0] +delta*cos(alpha),theta[1]+delta*sin(alpha)]

def neighbors(theta):
    neighbors_list = []
    cont=0
    alpha=2*np.pi/num_neighbors
    Enquanto cont <num_neighbors:
        neighbors_list.append([theta[0]+delta*cos(cont*alpha),theta[1]+ delta*sin(cont*alpha)])
        cont +=1
    retorne neighbors_list
```

3 Gradient Descent

```
def gradient_descent(cost_function, gradient_function, theta0, alpha, epsilon, max_iterations):

    theta = theta0
    history = [theta0]
    i=0
    Enquanto i<max_iterations e cost_function(theta)>=epsilon:
        theta=theta - alpha*gradient_function(theta)
        history.append(theta)
        i+=1
    retorne theta, history
```

Os resultados da implementação do *Gradient Descent* estão mostrados na figura 1.

4 Hill climbing

```
def hill_climbing(cost_function, neighbors, theta0, epsilon, max_iterations):
    theta = theta0
    history = [theta0]
    i=0

    Enquanto i<max_iterations e cost_function(theta)>=epsilon:
        best = None
        min_cost = infinito
        para neighbors in neighbors(theta):
            se cost_function(neighbor)<min_cost:
                best=neighbor
                min_cost=cost_function(neighbor)
```

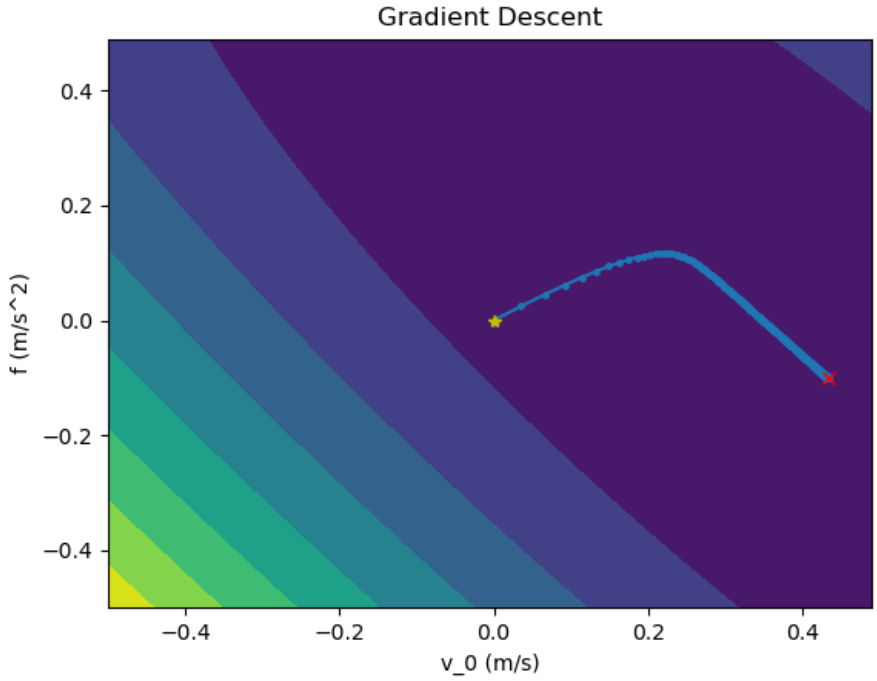

Figura 1: Resultado da otimização com o algoritmo descida do gradiente

```
        se min_cost>cost_function(theta):
            retorne theta, history
        theta= best
        history.append(theta)

        i+=1
    retorne theta, history
```

Os resultados da implementação do algorítimo *Hill Climbing* estão apresentados na figura 2.

5 Simulated Anealling

```
def simulated_annealing(cost_function, random_neighbor, schedule, theta0, epsilon, max_iterations)
    theta = theta0
    history = [theta0]
    i=0
    Enquanto i < max_iterations e cost_function(theta) >= epsilon:
        T=schedule(i)
        se T<0.0:
            retorne theta,history
        neighbor=random_neighbor(theta)
        deltaE=cost_function(neighbor)-cost_function(theta)
        se deltaE<0:
            theta=neighbor
        senão:
            r=random_uniform(0.0,1.0)
            se r>= exp(deltaE/T):
                theta=neighbor
        history.append(theta)
        i+=1

    retorne theta, history
```

Os resultados da implementaçao do algorítimo *Simulated Anealling* estão apresentados na figura 3.

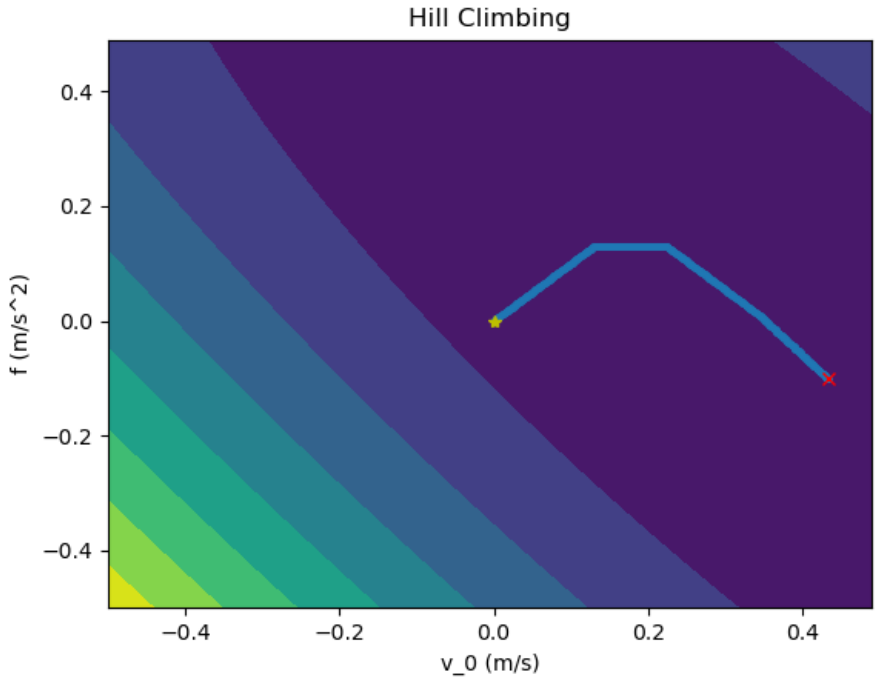

Figura 2: Resultado da implementação do algorítimo *hill climbing*

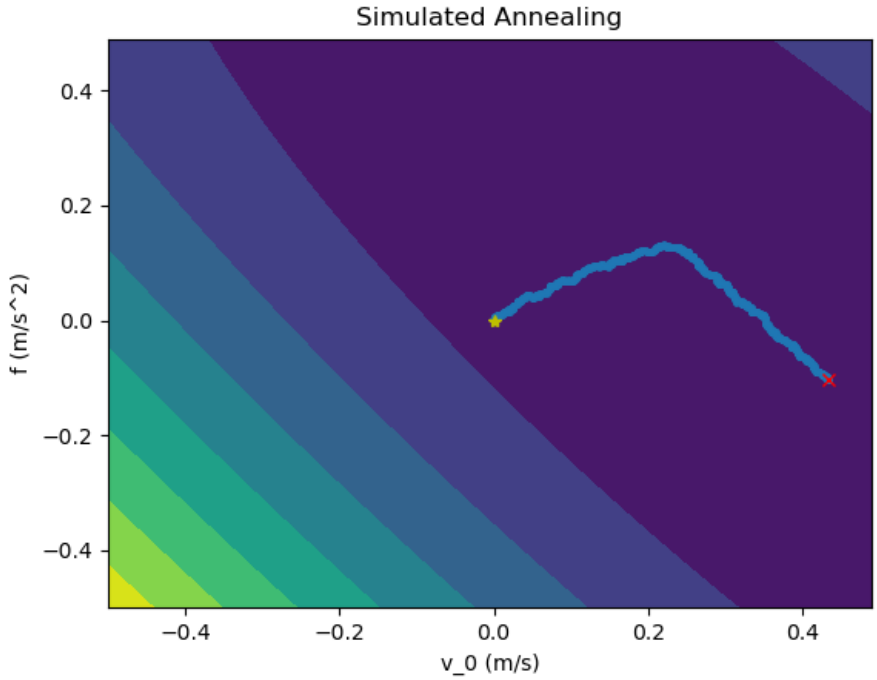

Figura 3: Resultado da implementação do Simulated Aneallling

6 Comparação dos resultados com o MMQ

Como apresentado pelo professor no roteiro do laboratório, tal problema tinha a melhor solução através do Método dos Mínimos Quadrados (MMQ). Pode-se observar na figura 4 que o algorítimo que melhor se comporta em tal problema e tais condições é o algorítimo da descida do gradiente

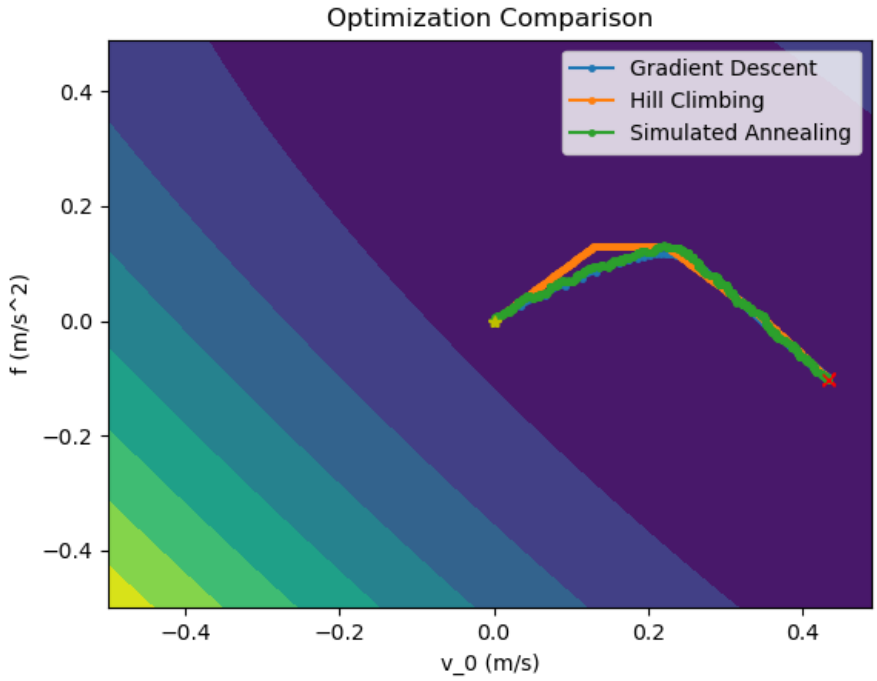

Figura 4: Comparação entre métodos de otimização