

Laboratório 4 – Otimização com Métodos Baseados em População

1. Introdução

Nesse laboratório, seu objetivo é implementar uma otimização dos parâmetros do controlador de um robô seguidor de linha usando *Particle Swarm Optimization* (PSO). A Figura 1 mostra a trajetória do robô após otimização com PSO. Note que a trajetória obtida por você pode ser diferente, pois sua execução da otimização pode convergir para outro máximo local.

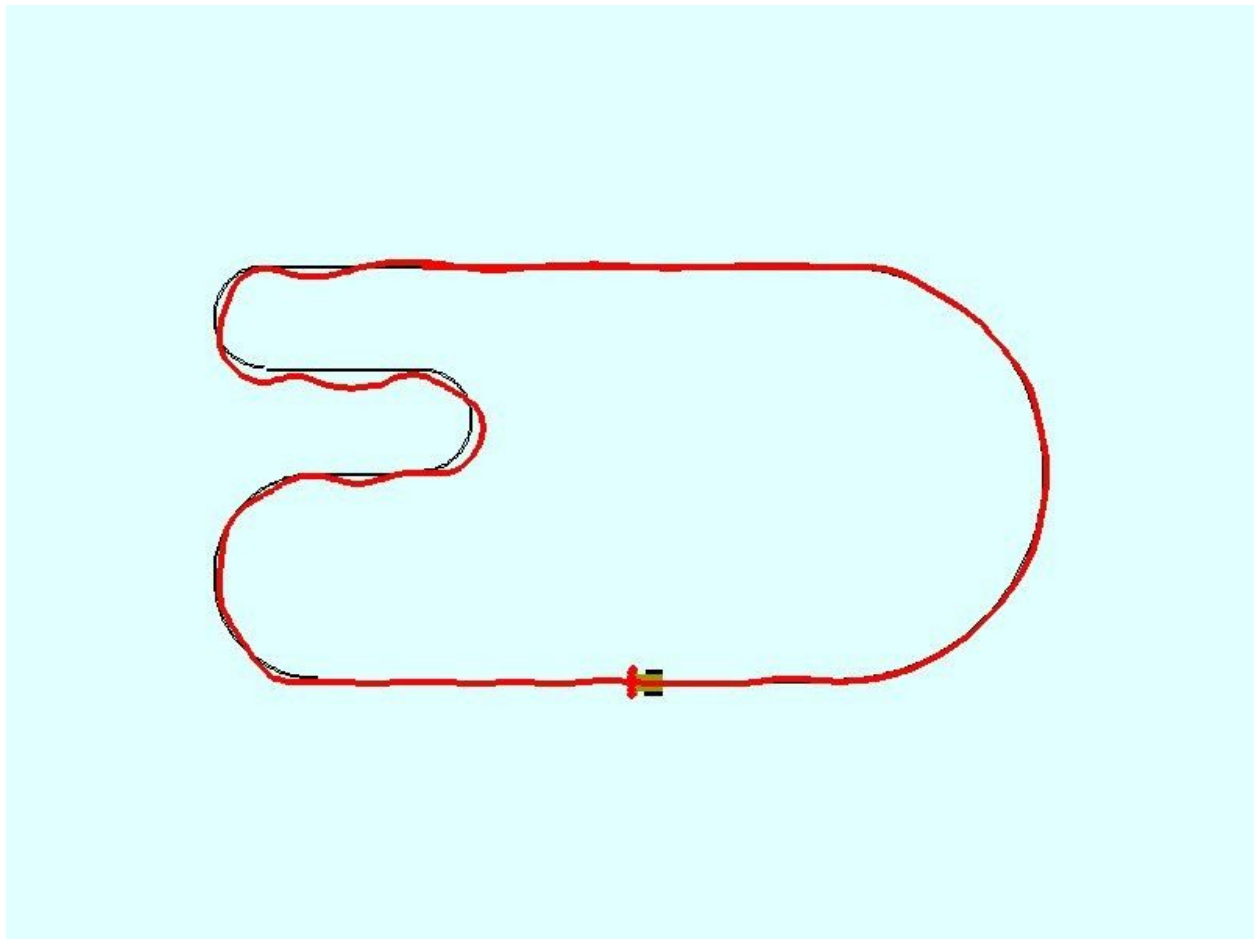


Figura 1: trajetória do robô seguidor de linha após otimização com PSO.

2. Descrição do Problema

O problema ser resolvido é a otimização do controlador de um robô seguidor de linha. O controlador mantém velocidade linear constante, enquanto utiliza um controlador PID para escolher a velocidade angular:

$$\omega = K_p e + K_i \int e dt + K_d \dot{e}$$

em que ω é a velocidade angular comandada e e é o erro de seguimento da linha. Além disso, K_p , K_i e K_d são os ganhos proporcional, integrativo e derivativo, respectivamente. Os parâmetros sujeitos a otimização são a velocidade linear e os 3 ganhos do controlador, logo são 4 parâmetros.

Como medida de qualidade, recomenda-se usar a seguinte equação:

$$f(x) = \sum_{k=1}^N reward_k$$

em que $reward_k$ é a “recompensa” obtida pelo robô no instante k e N é a duração do episódio de treinamento em passos de tempo (*time steps*). Para o cálculo da recompensa, sugere-se:

$$reward_k = v_k * dot(r_k, t_k) - w * |e_k|$$

em que v_k é a velocidade linear (projetada no eixo local do robô) executada pelo robô no instante k , r_k é um vetor (bidimensional) unitário que aponta na direção do robô no instante k , t_k é o vetor tangente à atual posição no caminho no instante k , $|e_k|$ é o módulo do erro em relação à linha e w é um peso para fazer um compromisso entre se manter no centro da linha e seguir o caminho rapidamente.

Perceba que essa medida de qualidade realiza recompensa o robô por cumprir o caminho mais rapidamente, enquanto o penaliza por desviar da linha. O termo $dot(r_k, t_k)$ significa o produto interno entre r_k e t_k , e foi adicionado para penalizar o robô caso ele se mova em reverso, pois sem esse termo acontecia de o PSO às vezes convergir para uma solução que o robô fazia o circuito (ou uma parte dele) em reverso (ao contrário). Perceba que quando o robô está perfeitamente alinhado com a linha, tem-se $dot(r_k, t_k) = 1$, enquanto quando ele se move em reverso, tem-se $dot(r_k, t_k) = -1$.

3. Código Base

O código base já implementa a simulação do seguidor de linha. Segue uma breve descrição dos arquivos fornecidos:

- `constants.py`: arquivo de constantes.
- `discrete_pid_controller.py`: implementa um controlador PID discreto.
- `line_follower.py`: implementa o robô seguidor de linha, tanto sua simulação quanto seu controlador.
- `low_pass_filter.py`: implementa um filtro de primeiro ordem discreto. Esse filtro é utilizado para simulador a dinâmica da roda do robô.
- `main.py`: arquivo principal, que roda a otimização do seguidor de linha.
- `particle_swarm_optimization.py`: implementação do algoritmo *Particle Swarm Optimization* (PSO).

- `simulation.py`: implementa a simulação do robô seguidor de linha.
- `test_pso.py`: teste com função simples para verificar o correto funcionamento do PSO.
- `track.py`: classe que representa o circuito de linha.
- `utils.py`: diversas classes e métodos utilitários.

O foco da sua implementação nesse laboratório são os arquivos `particle_swarm_optimization.py` e `simulation.py`. Perceba que no `simulation.py`, você apenas precisa se preocupar com o método `evaluate()`, que calcula a recompensa dada a situação atual da simulação. Além disso, é interessante usar o arquivo `test_pso.py` para verificar sua implementação do PSO antes de tentar realizar a otimização do seguidor de linha.

4. Tarefas

4.1. Implementação do PSO

Sua primeira tarefa é a implementação do algoritmo *Particle Swarm Optimization*. Para isso, implemente os métodos do arquivo `particle_swarm_optimization.py`. Algumas dicas:

- Sua implementação de PSO deve considerar **maximização**.
- A classe `Particle` foi pensada para ser apenas uma classe que armazena variáveis relacionadas a uma partícula. Assim, não se pensou em manter lógicas complexas nesta classe, além da criação da partícula em si.
- A interface com o algoritmo de PSO é um pouco diferente da mostrada nos slides, de modo que a avaliação da função medida de qualidade é feita externamente ao objeto. Essa forma de implementar foi escolhida por ser mais conveniente para o caso deste laboratório. O pseudocódigo do uso dessa classe é o seguinte:

```
for i in range(num_evaluations):
    position = pso.get_position_to_evaluate()
    value = quality_function(position)
    pso.notify_evaluation(value)
```

- O método `advance_generation()` é o responsável por avançar as partículas após todas as partículas terem sido avaliadas na geração atual.
- **Não** há necessidade de implementar o salvamento e carregamento do estado do algoritmo no disco (para ser capaz de recuperar o algoritmo caso ocorra alguma falha).

Então, teste sua implementação usando o arquivo `test_pso.py`. A função otimizada é:

$$f(x) = -((x(0) - 1)^2 + (x(1) - 2)^2 + (x(2) - 3)^2)$$

O ótimo global dessa função é $[1 \ 2 \ 3]^T$. Além disso, a função é convexa, de modo que não há outros ótimos locais. Nos meus testes, o PSO converge perfeitamente para o ótimo global, dado que é usado um número de 1000 gerações com 40 partículas. Obviamente, só é

possível executar tantas gerações pois a função medida de qualidade nesse caso tem um custo computacional muito baixo. Inclua no seu relatório os gráficos gerados por esse teste.

4.2. Otimização do controlador do robô seguidor de linha

A otimização do controlador do robô seguidor de linha é realizada através do arquivo `main.py`. Enquanto o PSO não tiver sido implementado, o robô não irá se mexer, pois a implementação inicial “falsa” do PSO está sempre passando velocidade linear nula como parâmetro do controlador. Além da execução da simulação em si, o código base fornece algumas opções de interação com o usuário através de teclas do teclado:

- A: ativa/desativa o modo acelerado (*accelerated mode*). Quando em modo acelerado, a simulação é executada até 200x mais rápido que tempo real (depende também da capacidade do seu computador). Você também pode controlar esse fator através das setas do teclado (para cima/para baixo aumenta/diminui em 1x, enquanto para direita/para esquerda aumenta/diminui em 10x).
- T: ativa/desativa o treinamento do robô. Se o treinamento estiver desativado, o robô usa os melhores parâmetros encontrados pelo PSO até então. Além disso, se o treinamento estiver desativado, salva uma captura da tela no fim do episódio.
- P: plota gráficos do histórico do treinamento. Além disso, salva estes gráficos em formato .png para serem incluídos no relatório.

Além da implementação do PSO, é necessário implementar a recompensa r_k usada na medida de qualidade do problema (o acúmulo dos r_k num dado episódio já está implementado). Isso deve ser feito no método `evaluate()` da classe `Simulation` (do arquivo `simulation.py`). Para facilitar, o método foi entregue parcialmente implementado, de modo que você não precise entender o restante do código para terminar a implementação. Porém, atente-se para os seguintes fatos:

- **Perceba que o método `LineSensorArray.get_error()` retorna 0 como erro caso a linha não seja detectada.** Assim, teste se a linha foi detectada (booleana `detection`) e coloque algum valor padrão para `error` caso não tenha sido. Tem que ser algum valor maior que o máximo erro detectável pelo sensor: 0,03 (que é metade da largura do sensor).
- O valor de w na função medida de qualidade é meio arbitrário (quanto maior w , espera-se que o robô encontre soluções mais “cautelosas”). $w = 0,5$ funcionou bem nos meus testes.

Lembre de sempre rodar a otimização em modo acelerado para evitar esperar tanto. Nos meus testes, com cerca de 20 minutos (de tempo de relógio), a otimização já tinha encontrado boas soluções, próximas da ótima. Não há necessidade de esperar o PSO convergir para os propósitos deste laboratório, o que demorou no mínimo 4 horas nos meus testes. Assim, adote 3000 iterações (avaliações de medida de qualidade) como suficiente para os propósitos deste laboratório.

Finalmente, destaco que mantive no código o circuito “simples” para caso alguém tenha muita dificuldade em otimizar no circuito “complexo”. Para fazer essa troca, troque a linha 221

para `track = create_simple_track()`". Por favor, fale comigo antes de entregar o lab caso precise fazer isso.

Inclua no seu relatório gráficos do histórico de otimização (os que são plotados quando se aperta P) e a melhor trajetória obtida durante a otimização (salva quando se desativa o treinamento). Discuta também o que você observou durante o processo de otimização.

A propósito, antes que me perguntem: o circuito não está quebrado nas curvas! Eu confesso que tentei arrumar isso, mas a pygame desenha arcos desse jeito mal-feito mesmo. O que importa mais é que a simulação que roda por debaixo está correta.

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 4 como "**marcos.maximo_lab4.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 de zeros usando NumPy que tenha o mesmo número de elementos que outro vetor, faça:

```
array = np.zeros(np.size(other_array))
```

- Operações normais de vetor, como soma, subtração e multiplicação por escalar, funciona como esperado em NumPy:

```
sum = a + b
```

```
sub = a - b
```

```
mul_scalar = scalar * a
```

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

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

- Você também pode amostrar um vetor aleatório usando NumPy:

```
array = np.random.uniform(lower, upper)
```