

Optimization and Learning for Robot Control: Final Project

Luca Sartore - 256154

Contents

1	Introduction	2
2	Creating a dataset using Optimal Control	2
2.1	Solving the optimal control problem	2
2.1.1	Optimal control problem formulation	2
2.1.2	Optimization issues	3
2.2	Dataset creation	4
3	Training the critic	4
3.1	Simple Critic	5
3.2	Inertia Critic	5
4	Training the actor	6
4.1	Simple Actor	6
4.2	Inertia Actor	7
5	Optimal control vs Actor Approximation	7
5.1	Simple system trajectories	7
5.2	Inertia system trajectories	9
6	Solving time of different methods	11
6.1	results	11
6.1.1	Simple system plots	12
6.1.2	Inertia system plots	13

1 Introduction

This work aims to develop a neural network that approximates the output of an optimal control problem.

The selected system is straightforward: a robot moves in one dimension. At each iteration the robot receives a cost that is proportional to the squared control input and to a positional cost. The objective of the robot is to plan a trajectory with the lowest possible cost.

The tests were made with two systems. One is named “Simple”, where the robot’s velocity is proportional to the control input, and the other is named “Inertia”, where the robot has inertia and the control input therefore becomes an acceleration.

The system is trained using a Critic-Actor paradigm, and the results are evaluated in two main ways:

- **Trajectory:** Here the optimal trajectories are compared with the trajectories obtained by the neural network
- **Computation time** In this test we compare the computational cost of the various approaches used to solve the same problem

2 Creating a dataset using Optimal Control

2.1 Solving the optimal control problem

The first step in the creation of the dataset, was to solve an optimal control problem. And the first step to do so is to formulate the optimal control problem

2.1.1 Optimal control problem formulation

In the optimal control problem, u is the input (in this case one-dimensional) to the system, while x is the state variable and its dimensionality depends on the system tested.

$$x = \begin{cases} \begin{bmatrix} p \end{bmatrix} & \text{for “simple” system} \\ \begin{bmatrix} p \\ v \end{bmatrix} & \text{for “inertia” system} \end{cases}$$

where:

p is the position

v is the velocity

Then the optimal control problem can be defined in simple terms with the equation below.

Notable components are l (the cost function), which depends on the squared input plus a positional cost, and f , which represents the dynamics of the system (and is therefore

dependent on the selected system).

$$\begin{aligned}
& \underset{X, U}{\text{minimize}} && \sum_{i=0}^{N-1} l(x_i, u_i) \\
& \text{subject to} && x_{i+1} = f(x_i, u_i) \quad i = 0 \dots N-1 \\
& && x_0 = x_{init} \\
& \text{where} && \\
& && l(x, u) = \frac{1}{2} * u^2 + (p - 1.9)(p - 1.0)(p - 0.6)(p + 0.5)(p + 1.2)(p + 2.1) \\
& && f(x) = \begin{cases} \begin{bmatrix} p \end{bmatrix} + \begin{bmatrix} p + \delta t \cdot u \end{bmatrix} & \text{for "simple" system} \\ \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} p + \delta t \cdot v + \frac{1}{2} \Delta t^2 \cdot u \\ v + \Delta t \cdot u \end{bmatrix} & \text{for "inertia" system} \end{cases}
\end{aligned}$$

2.1.2 Optimization issues

During the optimization a few issues emerged, and the solutions are reported below.

The first issue was that the solver was biased towards the cost minimum that is found near 0. As we can see in figure 1 from the initial position (-1.3) the solver should converge in -1.8 as it is both closer, and lower cost. However the solver prefer to “walk uphill” the cost’s gradient and end up in a worst spot.

This is because no initial guess was provided, and the state variables were therefore initialized at zero, making the solver biased.

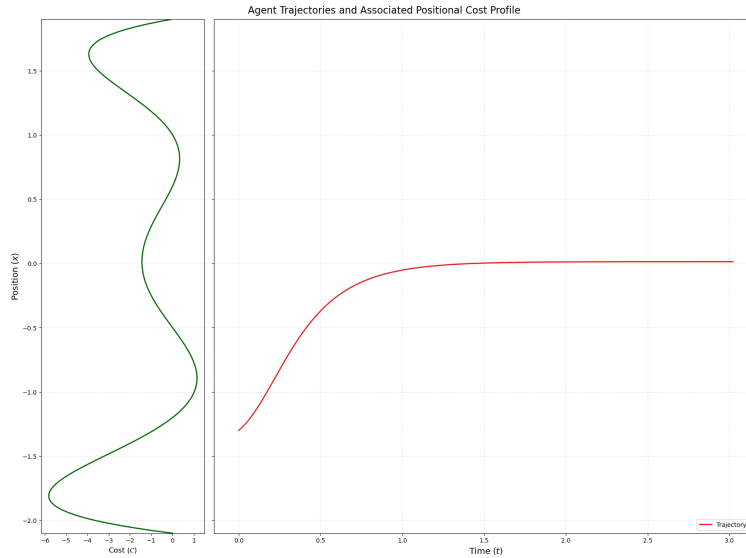


Figure 1: An example of how the solver prefers a suboptimal solution found near 0 to a better one found near -1.8

To address this issue the state variables were initialized with random values within a

specific range. This made it more likely that the solver selected a different minimum than the one at zero. However, there was still high variability in where the solver would end up (even with the initial state being exactly the same). To create a more accurate dataset, each input point was solved 10 times and only the best solution was added to the dataset.

2.2 Dataset creation

The dataset creation was straightforward. The inputs were generated by picking uniform samples within the following ranges:

- -2.15 to 1.95 for the position
- -5 to 5 for the position velocity

The dataset size was selected to be 10k (meaning that 100k OCP problems were solved). The calculation required approximately 45 minutes (for each of the two systems tested) on a 6-core Ryzen 5 5600 CPU. The implementation was multithreaded.

3 Training the critic

The training of the critic was done using the following parameters:

- **Loss:** L1 loss (L2 and Huber were also tested, but found to be worse)
- **Optimizer:** AdamW with a learning rate of 10^{-4}
- **LR Scheduler:** The learning rate was scheduled so that it was cut in half every time there were 10 iterations without any improvement.
- **Train-Test split:** The dataset was split 90/10 for training and validation
- **Batch size:** 64 (other options were tested, but the difference seemed minimal)
- **Iterations:** Max iterations were set to 2000 (even though the algorithm was often terminated after about 500 iterations for lack of improvements).

The final validation losses work out to be around 1.0 and 3.3 for the “simple” and “inertia” systems respectively, while the training time was approximately 1 minute 45 seconds on an RTX 3060 Ti GPU.

The sizes of the networks were chosen by gradually increasing the size of the various layers until the improvement was marginal. A similar logic was applied to the number of layers. A “shrinking” architecture (where the layers start wide and shrink as they get closer to the output) was selected, as is common in regression problems.

3.1 Simple Critic

The network for the “simple” system’s critic ended up being a 4-layer FFNN with the respective layer widths (1,1024,256,1). All the activation functions are leaky ReLU.

In Figure 2 we can see how accurate the critic is compared to the ground truth.

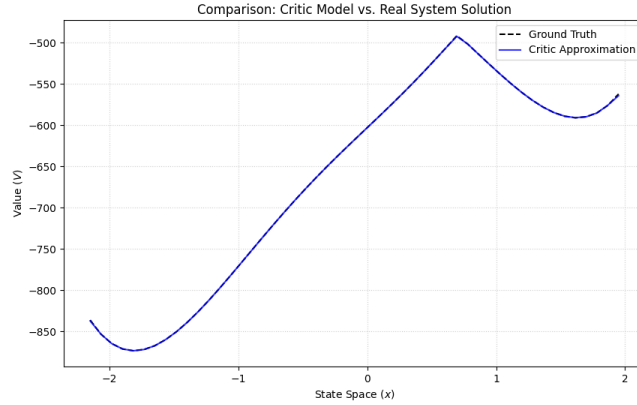


Figure 2: Simple Critic accuracy compared to ground truth

3.2 Inertia Critic

The network for the “inertia” system’s critic ended up being a 5-layer FFNN with the respective layer widths (2,1024,512,256,1). All the activation functions are leaky ReLU.

In Figure 3 we can see how accurate the critic is compared to the ground truth.

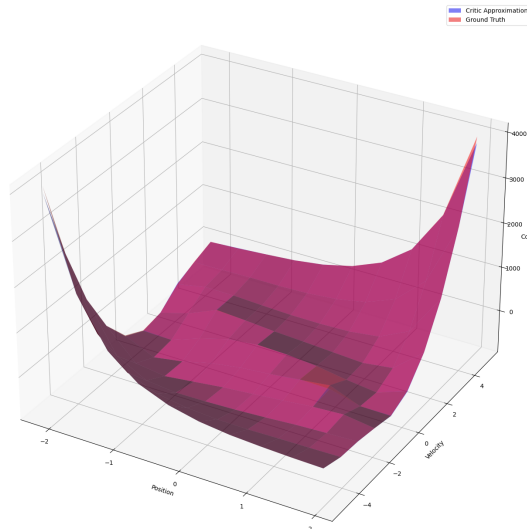


Figure 3: Inertia Critic accuracy compared to ground truth

4 Training the actor

The actor’s training was done by applying the following loss

$$Loss = l(x, \pi(x)) + V(x, f(x, \pi(x)))$$

Where l is the cost function, V is the value approximation (i.e. the critic), and π is the trained policy (i.e. the actor). When training with this loss, the weights of the critic are kept frozen while only the weights of the actor are optimized.

The parameters used during training are the following:

- **Optimizer:** AdamW with a learning rate of $5 \cdot 10^{-5}$
- **Batch size:** 2048 (other batch sizes were tested, but the difference was minimal)
- **Iterations:** 5000 (where each iteration had only one batch with the input state being randomly generated)

Initially the training process did not work at all; the reason turned out to be interesting. During training the state ended up outside the range considered ”valid” for the critic, and therefore the output was inconsistent. To solve this issue the critic was modified so that the input is clipped if it falls outside the validity range.

The training took approximately 11 seconds on an RTX 3060 Ti GPU. The final losses are not reported here as they are hard to interpret (given that they depend on the value function). The size and shape of the network were selected in the same way the critic’s were.

4.1 Simple Actor

The network for the “simple” system’s actor ended up being a 4-layer FFNN with the respective layer widths (1,512,128,1). All the activation functions are leaky ReLU.

In Figure 4 we can see how accurate the actor is compared to the ground truth.

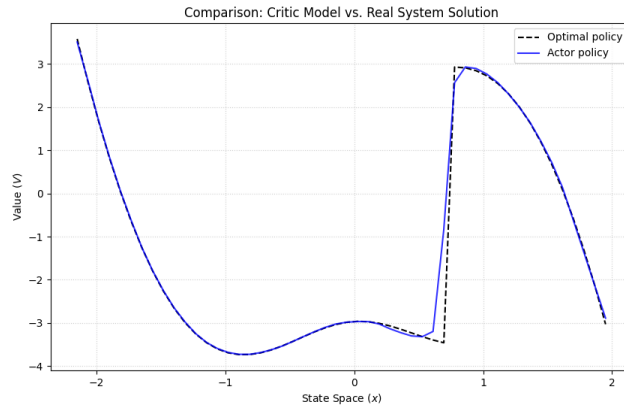


Figure 4: Simple Actor accuracy compared to ground truth

One interesting detail to note in the chart is how the system struggles a bit in the “discontinuity” found around 0.7. The discontinuity is due to the fact that around that value the optimal strategy changes from moving towards the minimum around -1.8 to moving to the minimum around 1.8 .

This is potentially solvable using larger networks, but that would undermine the main advantage of this approach, which is speed, and would also potentially require a larger dataset.

4.2 Inertia Actor

The network for the “inertia” system’s actor ended up being a 5-layer FFNN with the respective layer widths (1,1024,256,64,1). All the activation functions are leaky ReLU.

In Figure 5 we can see how accurate the actor is compared to the ground truth.

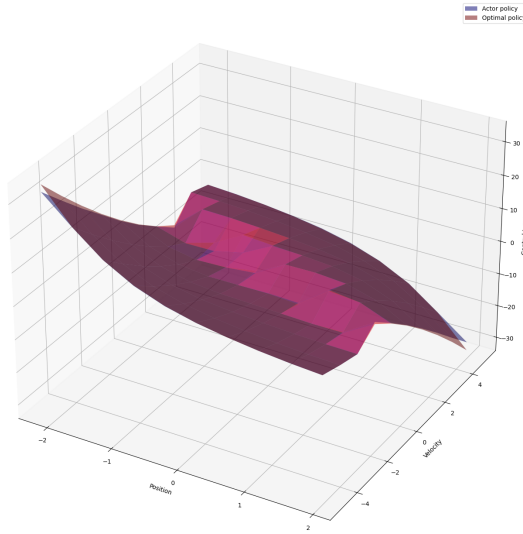


Figure 5: Inertia Actor accuracy compared to ground truth

5 Optimal control vs Actor Approximation

In this section we report some trajectories that have been obtained with optimal control and with the actor to show how well the network was able to learn the optimal policy.

5.1 Simple system trajectories

We can note that the trajectories look generally good. In the first sub-figure of Figure 6 there is a small divergence from the steady state reached by the two control methods.

This can be explained by looking at Figure 7, which shows how the critic’s value function has a local minimum that is slightly shifted with respect to the true value function. An issue is reflected in the learned policy.

Initially this issue was much larger and was reduced by increasing the size of the critic network. This stopped helping once the dataset size became the limiting factor.

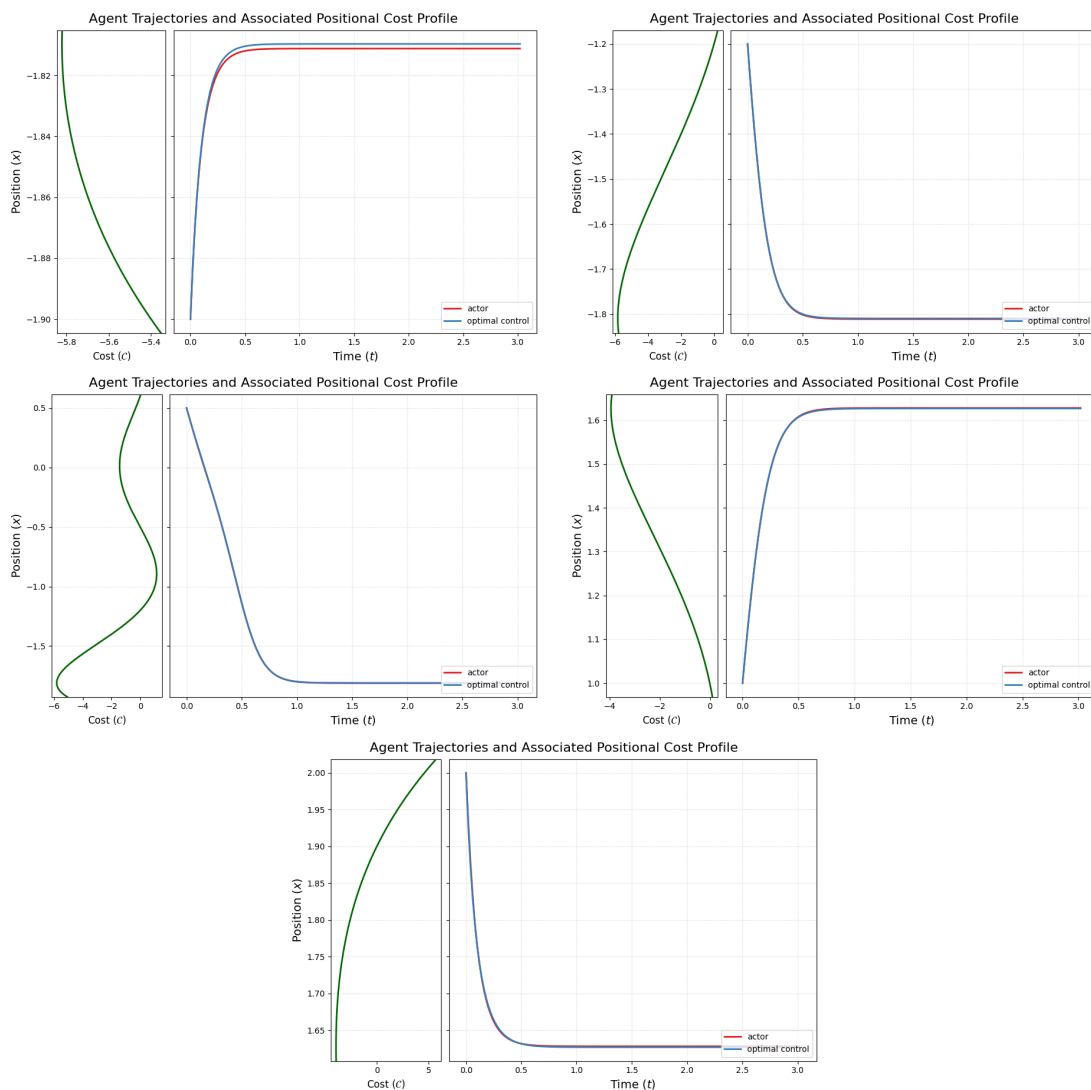


Figure 6: Trajectory in the “simple” system

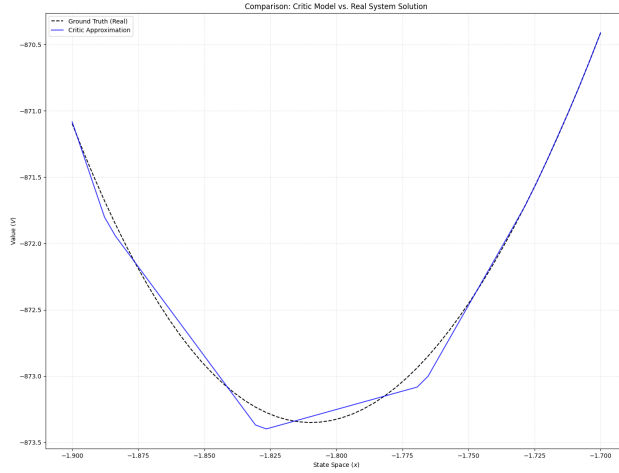


Figure 7: Value function around -1.8

5.2 Inertia system trajectories

The inertia system trajectories also look good and are in general consistent with those calculated using optimal control.

The first subplot of Figure 8 is the worst result, which makes intuitive sense as it is one of the most complex cases (where the initial position is close to the minimum cost but the robot has a high velocity in the opposite direction).

It is also worth noting that the closer the robot gets to the end, the more the behavior of the optimal control solver diverges from that of the actor. This is because the actor takes one action at a time and is trained to consider a full-length horizon for each action. Meanwhile, the optimal control solver solves the problem once, and the closer it gets to the horizon end the shorter the considered horizon becomes.

This effect can be noted in the first subplot of Figure 8, where near the end the optimal control trajectory is still moving downward. This is because the time horizon is about to end, and the cost of adjusting the trajectory is greater than the cost associated with drifting downward.

Meanwhile, the actor behaves as if it just started and still has a full horizon ahead; the adjustment cost is worth paying, resulting in a straighter trajectory.

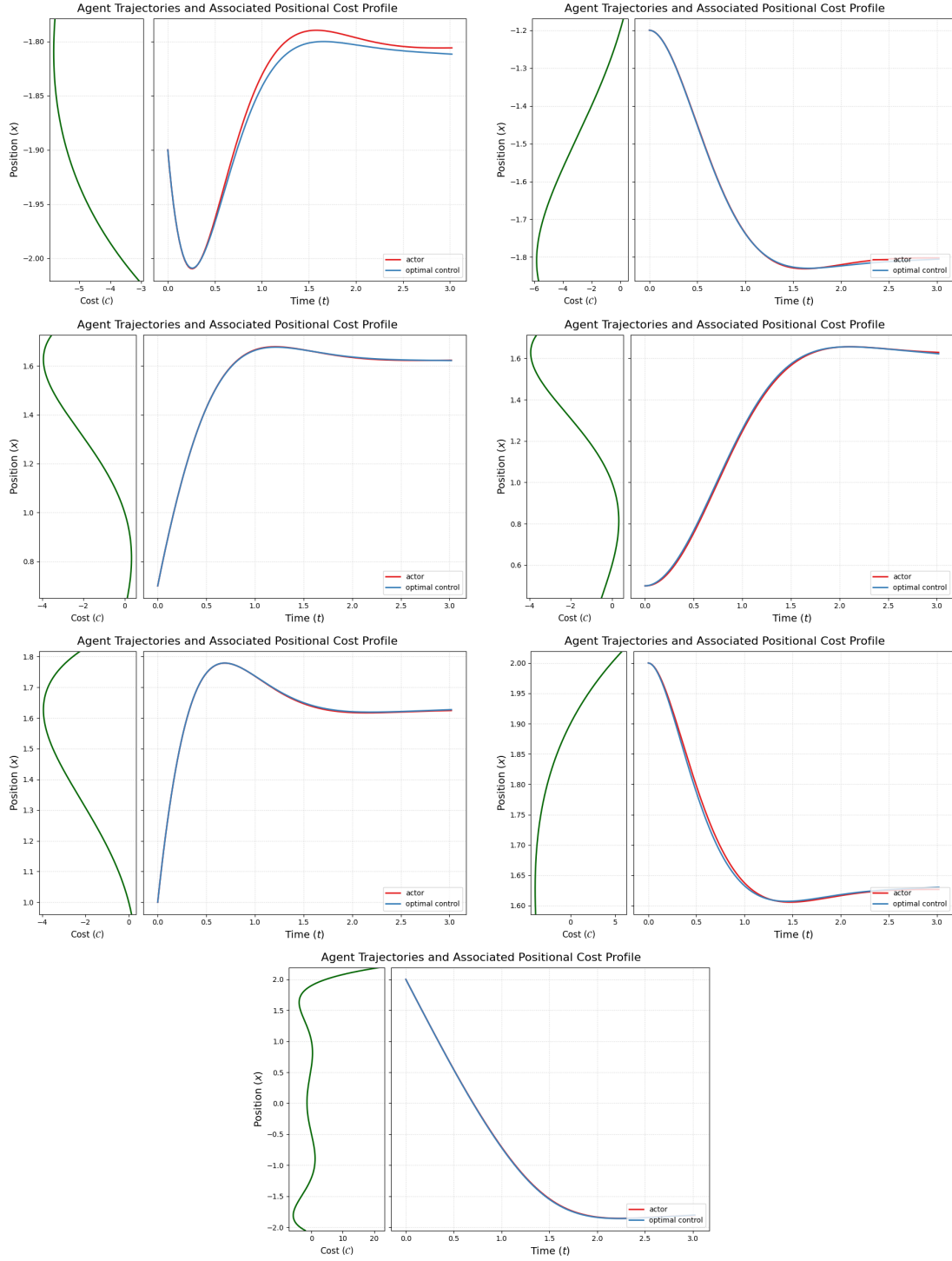


Figure 8: Trajectory in the “inertia” system

6 Solving time of different methods

The last step for this report is comparing four different solving methods to see execution times and trajectory accuracy. The considered systems are the following:

- **Optimal Control (best of one):** This method solves one optimal control problem and takes the output. It suffers from the fact that it sometimes finds solutions that are locally (but not globally) optimal (for the reasons described in section 2.1.2).
- **Optimal Control (best of ten):** This method solves 10 optimal control problems and picks the best solution. It is slower but does not suffer from the same issue as the previous option (or at least the likelihood of that issue occurring is much lower).
- **Actor:** This strategy simply involves using the actor to calculate the trajectory.
- **Optimal Control + actor initialization:** This method uses the actor to calculate an initial guess, and then solves one optimal control problem with initialized variables. This solution also doesn't suffer from the issue plaguing the first option.

6.1 results

The trajectories with execution times can be seen in figure 9 and 10. In the table below you can instead see a brief summary of how the various methods performed. In general we can say that the last two methods are the “pareto optimal” one.

The use of an initial guess in the optimal control did provide a tiny speed improvement to the solver, but this was canceled out by the time needed to run the actor, and the result is that the execution time is on par with a normal solver (or perhaps is even slightly worse, but it's within the margin of error.)

Strategy	Avg. exec. Time	Trajectory type	Local minimum issue
Optimal Control (best of one)	0.18 [s]	Optimal	Yes
Optimal Control (best of ten)	1.73 [s]	Optimal	No
Actor	0.03 [s]	Approximated	No
Optimal Control + actor initialization	0.18 [s]	Optimal	No

Table 1: Comparison between the tested solvers

6.1.1 Simple system plots

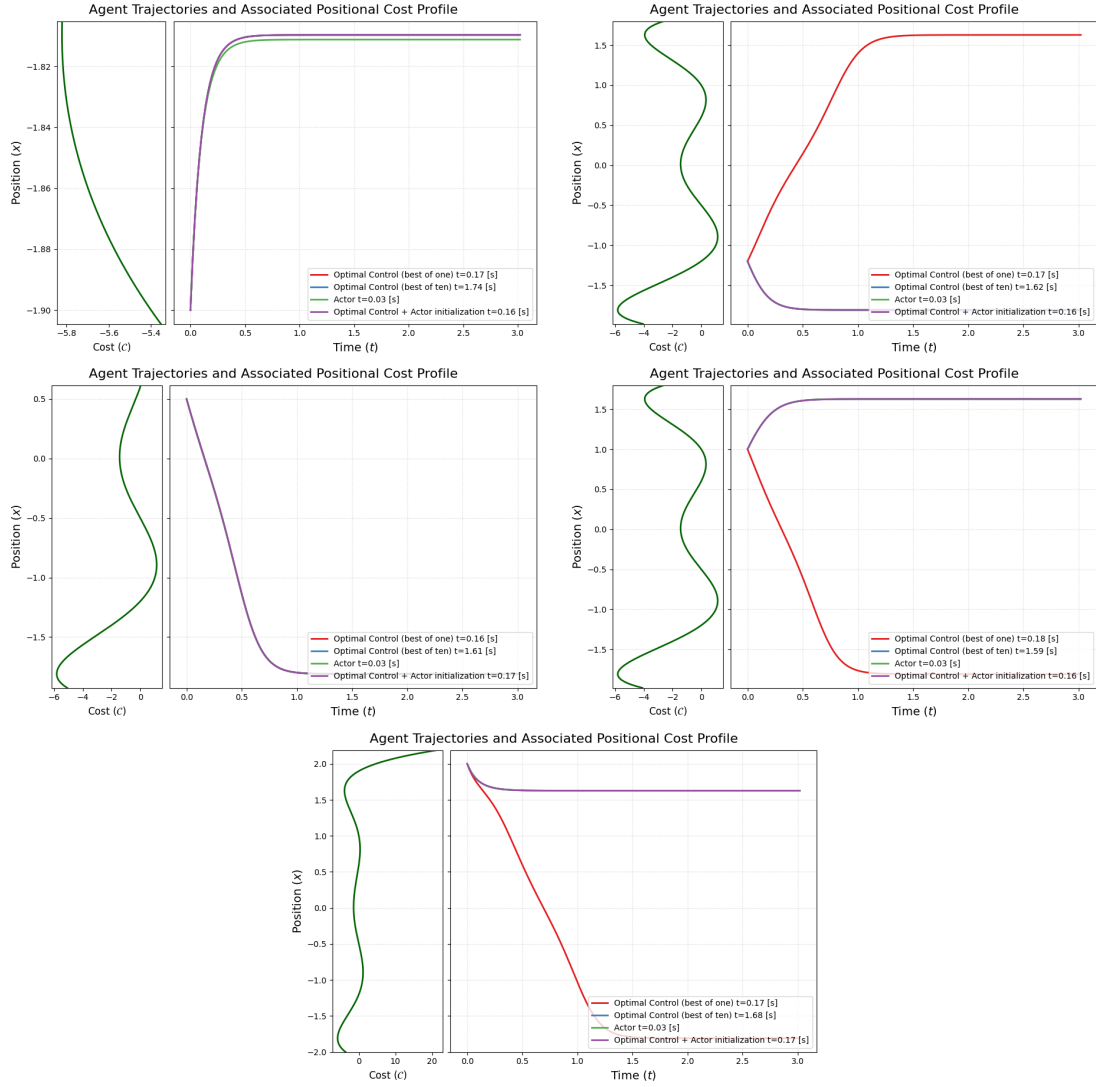


Figure 9: Different solution methods compared in the “simple” system

6.1.2 Inertia system plots

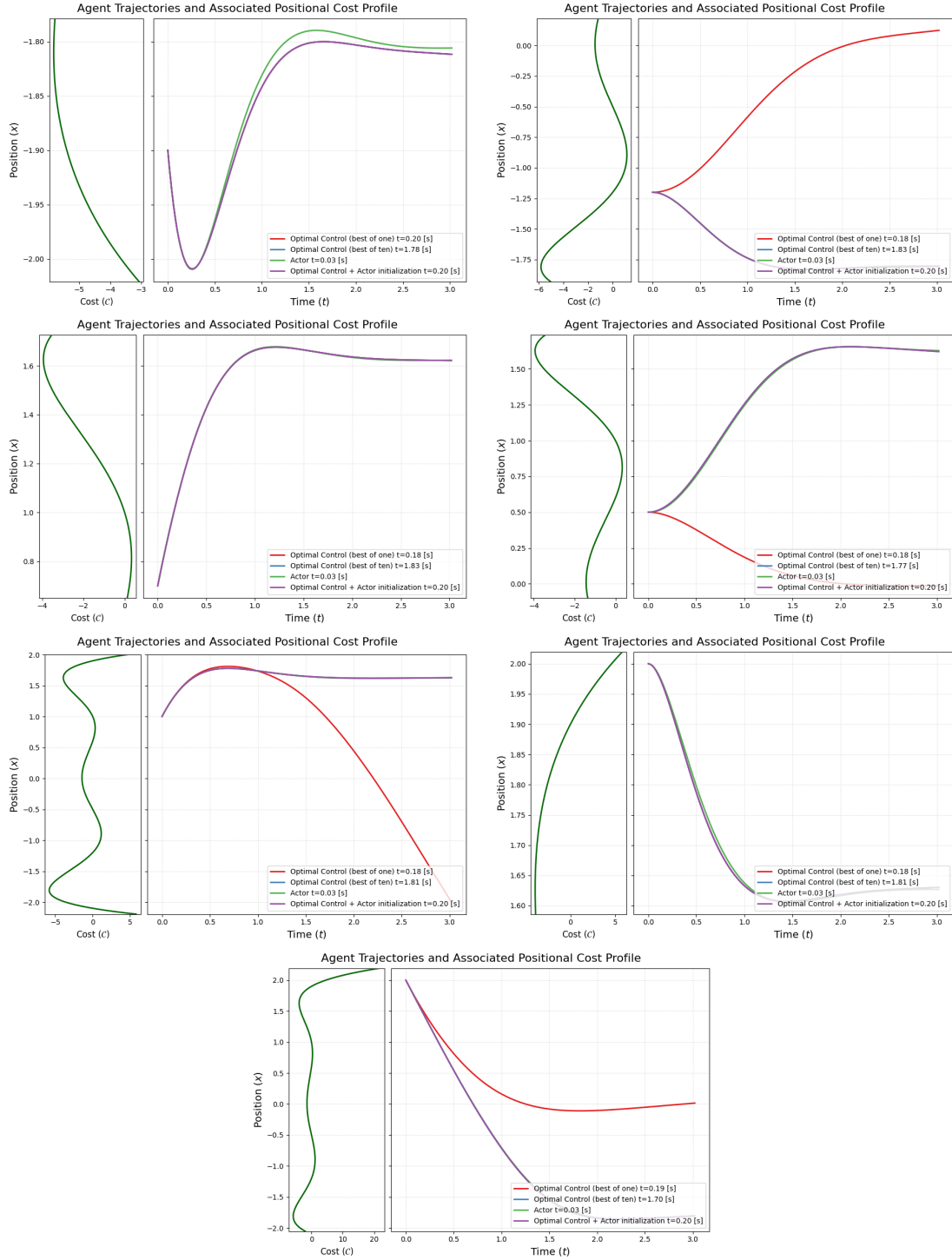


Figure 10: Different solution methods compared in the “inertia” system