ORC Assignment 3

Saviane Arianna mat. 240602

Titton Giulia mat. 240311

January 2024

Professor: Del Prete Andrea

Assistant Professor: Alboni Elisa

Contents

1 OCP with single integrator **1**
1.1 Results and plots 1

2 Neural network "critic" **2**
2.1 Critic's structure 3
2.2 Critic's code and training 4
2.2.1 Training, validation and testing 4
2.3 Results 5

3 Neural network "actor" **6**
3.1 Actor's structure 7
3.2 Actor's code and training 8
3.3 Results 8

4 Testing actor performance **9**
4.1 Actor's predictions as initial guess 9
4.2 Actor's predictions as direct control 13

5 OCP with double integrator **17**

6 Neural network "critic" in double integrator case **21**
6.1 Results 21

7 Neural network "actor" in double integrator case **22**
7.1 Results 23

8 Actor performances in double integrator case **24**
8.1 Actor's predictions as initial guess 24
8.2 Actor's predictions as direct control 31

9 Conclusions **37**

1 OCP with single integrator

This section describes the reasoning and the computation of the OCPs starting from different initial states and using a 1D single integrator to model the system.
The dynamics of the system can be modelled as:

$$x_{t+1} = x_t + \Delta t * u_t \tag{1}$$

while the running cost depends both on current state and current control, resulting in a non-convex problem:

$$l(x, u) = \frac{1}{2} * u^2 + (x - 1.9)(x - 1.0)(x - 0.6)(x + 0.5)(x + 1.2)(x + 2.1) \tag{2}$$

Many OCPs have been solved starting from different initial states (20000 in this case) ranging from a minimum of -2.2 to a maximum of 2 dividing the whole range in a vector containing the initial states equally spaced from each other. Controls can assume values ranging from -5 to 5. CasADi has been exploited to solve the OCPs explicitly extracting cost and optimal control of the OCPs for each and every initial state and saving the results into a *.npz* file.

1.1 Results and plots

The resultant plot of the computed OCPs cost is shown in Figure 1.

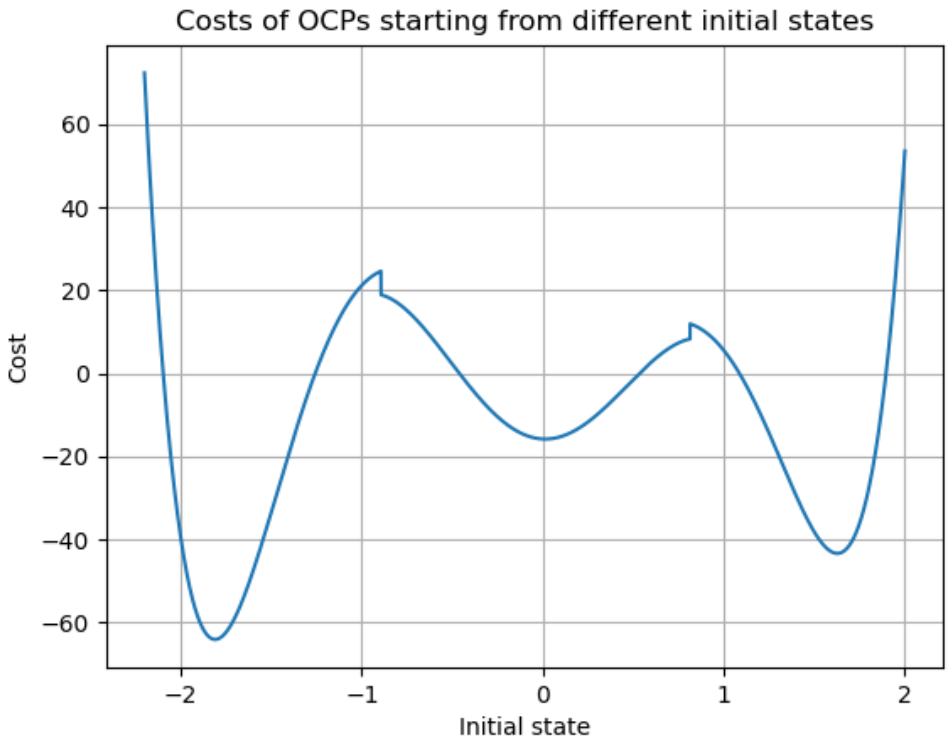


Figure 1: Cost of OCPs starting from different initial states.

Plotting the optimal control we get the behaviour in Figure 2.

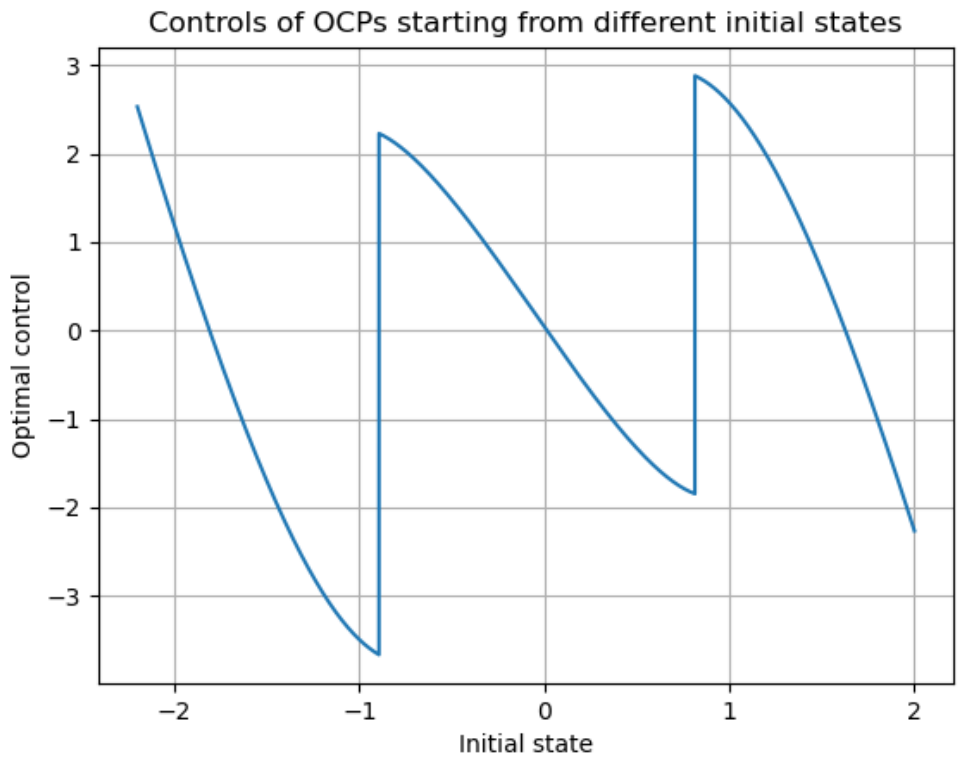
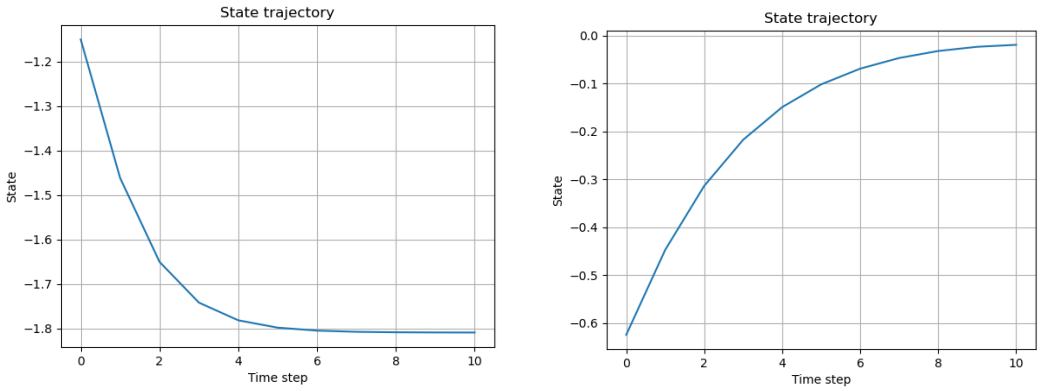


Figure 2: Optimal control of OCPs starting from different initial states.

Figure 3 shows two examples of state trajectories computed by the OCP solver.



(a) State trajectory starting from -1.15. (b) State trajectory starting from -0.625.

Figure 3: State trajectory examples.

2 Neural network "critic"

This section explains the training of a neural network called "critic" aimed at predicting the cost V given an initial state x_0. The inputs are taken from a file with

format *.npz* which stores the cost of the OCP for each initial state (x_0, V), as explained in the previous section.

2.1 Critic's structure

The neural network has the following structure:

- Input layer: composed of 1 neurons corresponding to x_0
- Hidden layer 1, with 16 neurons
- Hidden layer 2, with 32 neurons
- Hidden layer 3, with 32 neurons
- Output layer: 1 neuron which stores the result of the cost V

The activation function chosen for each layer is the ReLu, Rectified Linear Activation Unit. This function is the simplest and returns the value provided as input if the input is greater than zero, else it returns zero.
Regarding the loss function, instead, the mean squared error has been chosen in order to have an estimation of the loss between the target value and the predicted value from the network.

To be noted that other structures have been taken into account, but have been discarded since the results were not accurate and the model was not sufficiently complex for the network to learn a pattern. It has been observed that a network that expands the number of neurons in the layers is more suitable in this case than a network that decreases the number of neurons going from a layer to the next. For example the following structures have been considered:

- Input layer: composed of 1 neurons corresponding to x_0
- Hidden layer 1, with 16 neurons
- Hidden layer 2, with 8 neurons
- Hidden layer 3, with 8 neurons
- Output layer: 1 neuron which stores the result of the cost V

and

- Input layer: composed of 1 neurons corresponding to x_0
- Hidden layer 1, with 16 neurons
- Hidden layer 2, with 8 neurons
- Output layer: 1 neuron which stores the result of the cost V

The network structure explained above is the result of a trial and error process. It means that several trials have been made in order to tune the number of layers and the number of neurons for each layer, taking as reference the most popular network structures in machine learning. This result has been achieved trying to reach convergence and stability in epoch loss in order to balance overfitting and underfitting of data and to get the minumum error in prediction possible.

2.2 Critic's code and training

The code developed for the training of the neural network uses Tensorflow and its built-in functions, such as model.compile(), model.fit() and model.predict(). The training parameters are chosen as follows:

- Batch size = 32
- Epochs = 40
- Learning rate = is automatically chosen by the built-in functions in Tensorflow

More precisely, the batch size is an hyperparameter that represents the number of data samples processed in one iteration of the neural network. Choosing an appropriate batch size depends on several factors, including the size of the dataset, available memory, computational resources, and the specific characteristics of the problem.
The epochs parameter, instead, represents the number of times the entire dataset is passed forward and backward through the neural network during the training process. Using too few epochs may result in the model not learning enough from the data, leading to underfitting, i.e. poor performance on both training and unseen data. Conversely, using too many epochs might lead to overfitting, where the model performs well on the training data but fails to generalize to new, unseen data.
The size of the batch and the number of epochs have been chosen with the same aim of the choice of the number of layers and neurons. We saw that increasing the number of epochs did not give more accurate results, since after 30 to 40 iterations the values have already converged with minimun error. Instead, increasing or decreasing the batch size influenced only the computation time and not the results. Hence the parameter has been chosen in order to minimize the computation time, respecting the "good practice" of taking the value as power of 2.

2.2.1 Training, validation and testing

The training process has been divided into three phases: training, validation and testing. For this reason, the whole dataset, composed of 20000 samples of initial states and the corresponding costs, has been divided into three sets:

- Training dataset: composed by the 70% of samples coming from the whole dataset of computed OCPs. This dataset is used in the first and most important phase of training. It means that it is fed to the network in order to let it learn a pattern from the data.

- Validation dataset: composed by the 15% of samples. It is useful to validate the model used in training, while still training the network.
- Test dataset: composed by the remaining 15% of samples. It is important to test the model in a set of unseen data in order to ensure that the neural network has not learnt by heart but it has acquired a real and useful pattern.

2.3 Results

Figure 4 reports the results of the training process, with particular attention to the epoch loss and prediction error. It is possible to see how the epoch loss converges fast to a value and remains stable while the validation loss decreases faster. It means that the model works well and it is suitable for the purpose. It can also be noticed that the last epochs don't show large improvements, confirming that the number of epochs has been chosen reasonably.

```
Epoch 1/40
438/438 [==============================] - 5s 7ms/step - loss: 492.3460 - mse: 492.3460 - val_loss: 393.2830 - val_mse:
393.2830
Epoch 2/40
438/438 [==============================] - 2s 5ms/step - loss: 332.4705 - mse: 332.4705 - val_loss: 322.1354 - val_mse:
322.1354
Epoch 3/40
438/438 [==============================] - 2s 4ms/step - loss: 193.5919 - mse: 193.5919 - val_loss: 41.5633 - val_mse: 4
1.5633
Epoch 4/40
438/438 [==============================] - 2s 4ms/step - loss: 23.2555 - mse: 23.2555 - val_loss: 5.2463 - val_mse: 5.24
63
Epoch 5/40
438/438 [==============================] - 2s 4ms/step - loss: 6.6996 - mse: 6.6996 - val_loss: 3.0314 - val_mse: 3.0314
Epoch 6/40
438/438 [==============================] - 2s 4ms/step - loss: 5.6789 - mse: 5.6789 - val_loss: 15.3993 - val_mse: 15.39
93
Epoch 7/40
438/438 [==============================] - 2s 4ms/step - loss: 10.7656 - mse: 10.7656 - val_loss: 0.9105 - val_mse: 0.91
05
Epoch 8/40
438/438 [==============================] - 2s 4ms/step - loss: 1.6110 - mse: 1.6110 - val_loss: 2.8756 - val_mse: 2.8756
Epoch 9/40
438/438 [==============================] - 2s 3ms/step - loss: 4.1735 - mse: 4.1735 - val_loss: 0.8925 - val_mse: 0.8925
Epoch 10/40
438/438 [==============================] - 1s 3ms/step - loss: 3.9958 - mse: 3.9958 - val_loss: 2.5715 - val_mse: 2.5715
```

(a) First 10 epochs.

```
Epoch 30/40
438/438 [==============================] - 2s 5ms/step - loss: 2.0387 - mse: 2.0387 - val_loss: 0.3534 - val_mse: 0.3534
Epoch 31/40
438/438 [==============================] - 2s 5ms/step - loss: 0.3756 - mse: 0.3756 - val_loss: 0.1327 - val_mse: 0.1327
Epoch 32/40
438/438 [==============================] - 2s 5ms/step - loss: 2.0772 - mse: 2.0772 - val_loss: 5.5888 - val_mse: 5.5888
Epoch 33/40
438/438 [==============================] - 2s 5ms/step - loss: 9.5310 - mse: 9.5310 - val_loss: 0.2513 - val_mse: 0.2513
Epoch 34/40
438/438 [==============================] - 2s 5ms/step - loss: 0.7487 - mse: 0.7487 - val_loss: 3.9444 - val_mse: 3.9444
Epoch 35/40
438/438 [==============================] - 2s 5ms/step - loss: 1.5540 - mse: 1.5540 - val_loss: 0.7356 - val_mse: 0.7356
Epoch 36/40
438/438 [==============================] - 2s 5ms/step - loss: 5.9736 - mse: 5.9736 - val_loss: 0.6329 - val_mse: 0.6329
Epoch 37/40
438/438 [==============================] - 2s 5ms/step - loss: 0.3978 - mse: 0.3978 - val_loss: 0.2111 - val_mse: 0.2111
Epoch 38/40
438/438 [==============================] - 2s 5ms/step - loss: 5.0497 - mse: 5.0497 - val_loss: 0.3220 - val_mse: 0.3220
Epoch 39/40
438/438 [==============================] - 2s 5ms/step - loss: 0.3722 - mse: 0.3722 - val_loss: 0.1091 - val_mse: 0.1091
Epoch 40/40
438/438 [==============================] - 2s 5ms/step - loss: 2.5667 - mse: 2.5667 - val_loss: 1.4665 - val_mse: 1.4665
```

(b) Last 10 epochs.

Figure 4: Critic training.

The test phase in an unseen set of data has reported good results, hence we made

a prediction on the whole dataset, which is reported in Figure 5, and the values of predictions are stored in a file in order to be accessible from the actor network, in Section 3. The plot does not show a smooth behaviour as in Figure 1, but presents some irregularities due to loss in prediction of the model.

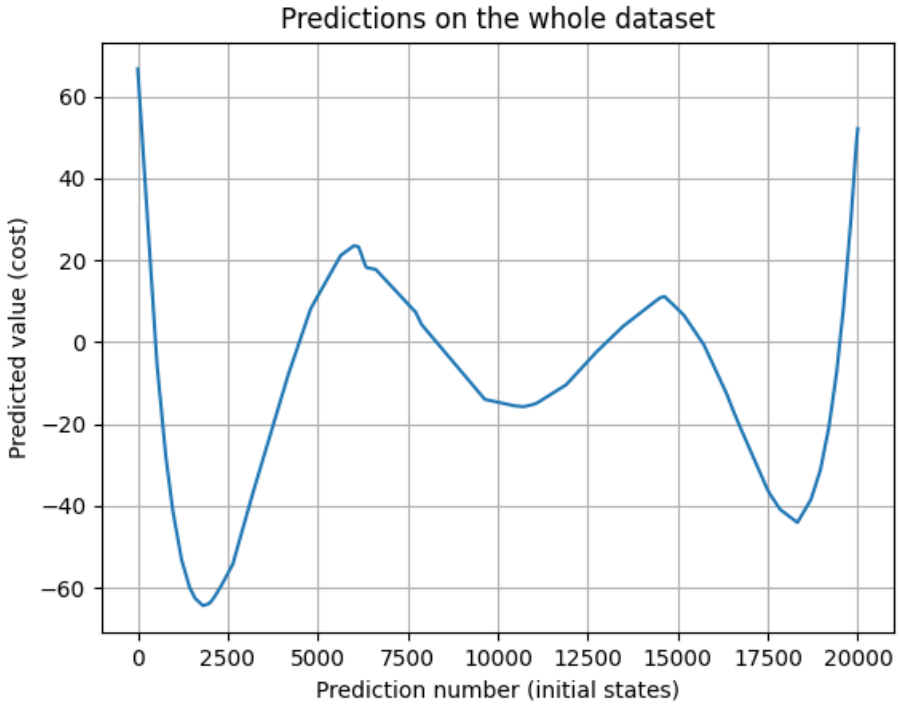


Figure 5: Predictions from the critic network.

The whole training process lasts 2 and a half minutes. The time taken is reasonably small considering the accuracy reached after training.

3 Neural network "actor"

Once the critic has been trained as in Section 2, the actor should be trained to approximate the greedy policy with respect to the critic.
Firstly, the action-value function corresponding to the critic should be minimized in order to find the policy π:

$$\pi(x) = \min_u \quad l(x,u) + V(f(x,u)), \tag{3}$$

where $l(x,u)$ is the running cost defined as in Equation 2, V is the critic and $f(x,u)$ is the next state calculated through the system dynamics starting from the current state and control. Discount factor has been taken equal to 1 to focus more on final cost rather than immediate cost. To perform the minimization, for each initial state a possible state at the next time step has been computed using the system dynamics as reported in Equation 1. The controls can assume values between -5 and 5 with a step size of 0.05 (200 possible controls). Consequently, for each initial state, the

greedy policy with respect to the critic has been found searching for the minimum value of the action value function depending on the control. The control related to the minimum value of Q is the greedy policy for that state. This phase takes 21 minutes, being the slowest part of the process. An array containing all the computed values for $\pi(x)$ has been saved in a *.npz* file and the results are shown in Figure 6.

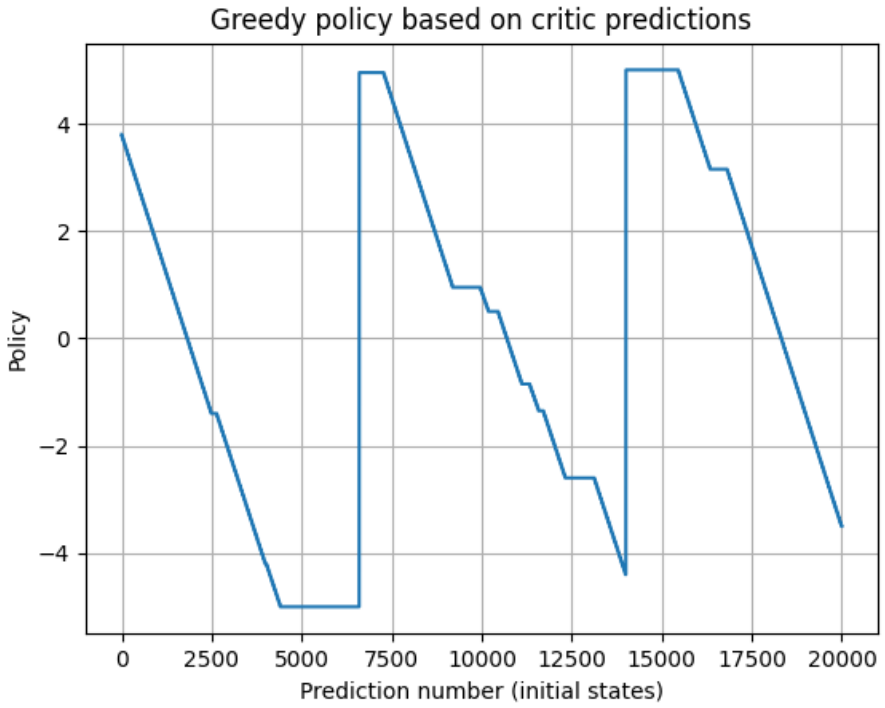


Figure 6: Greedy policy with respect to the critic.

3.1 Actor's structure

The neural network is built as follows, taking into account the complexity of the problem and the accuracy of results:

- Input layer: composed of 1 neuron corresponding to the greedy policy with respect to the critic.
- Hidden layer 1, with 16 neurons
- Hidden layer 2, with 32 neurons
- Hidden layer 3, with 32 neurons
- Output layer: 1 neuron which stores the approximation of the greedy policy.

Also in this case some other alternatives have been taken into account, but the best is the one described above, giving the best compromise between prediction loss and computation time.

3.2 Actor's code and training

As for the critic, also in this case the activation function chosen for each layer is the ReLu (Rectified Linear Activation Unit), while for the loss function the mean squared error has been chosen in order to have an estimation of the loss between the target value and the predicted value from the network.
The hyperparameters have been tuned as follows:

- Batch size = 32
- Epochs = 30
- Learning rate = automatically chosen by the built-in functions in Tensorflow

3.3 Results

Figure 7 reports the results of the actor training process, with particular attention to the epoch loss and prediction error. It is possible to notice that in the last epochs there are no large improvements and that additional epochs could not make the error decrease in a small amount of time.

```
438/438 [==============================] - 2s 3ms/step - loss: 0.3414 - mse: 0.3414 - val_loss: 0.2561 - val_mse: 0.2561
Epoch 16/30
438/438 [==============================] - 1s 3ms/step - loss: 0.3094 - mse: 0.3094 - val_loss: 0.2370 - val_mse: 0.2370
Epoch 17/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2912 - mse: 0.2912 - val_loss: 0.2958 - val_mse: 0.2958
Epoch 18/30
438/438 [==============================] - 1s 3ms/step - loss: 0.3007 - mse: 0.3007 - val_loss: 0.3943 - val_mse: 0.3943
Epoch 19/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2847 - mse: 0.2847 - val_loss: 0.2071 - val_mse: 0.2071
Epoch 20/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2792 - mse: 0.2792 - val_loss: 0.3046 - val_mse: 0.3046
Epoch 21/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2564 - mse: 0.2564 - val_loss: 0.2593 - val_mse: 0.2593
Epoch 22/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2507 - mse: 0.2507 - val_loss: 0.2565 - val_mse: 0.2565
Epoch 23/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2376 - mse: 0.2376 - val_loss: 0.2079 - val_mse: 0.2079
Epoch 24/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2632 - mse: 0.2632 - val_loss: 0.1742 - val_mse: 0.1742
Epoch 25/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2334 - mse: 0.2334 - val_loss: 0.2042 - val_mse: 0.2042
Epoch 26/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2607 - mse: 0.2607 - val_loss: 0.1932 - val_mse: 0.1932
Epoch 27/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2297 - mse: 0.2297 - val_loss: 0.1523 - val_mse: 0.1523
Epoch 28/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2607 - mse: 0.2607 - val_loss: 0.1776 - val_mse: 0.1776
Epoch 29/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2392 - mse: 0.2392 - val_loss: 0.2212 - val_mse: 0.2212
Epoch 30/30
438/438 [==============================] - 1s 3ms/step - loss: 0.2280 - mse: 0.2280 - val_loss: 0.1550 - val_mse: 0.1550
```

Figure 7: Actor training.

The test phase in an unseen set of data has reported good results, hence we made a prediction on the whole dataset, which is reported in Figure 8. The predicted values have been stored in a *.npz* file. The plot does not match exactly the values shown in Figure 6, but presents some irregularities due to loss in prediction of the model and the complex shape of the greedy policy, which does not behave smoothly.

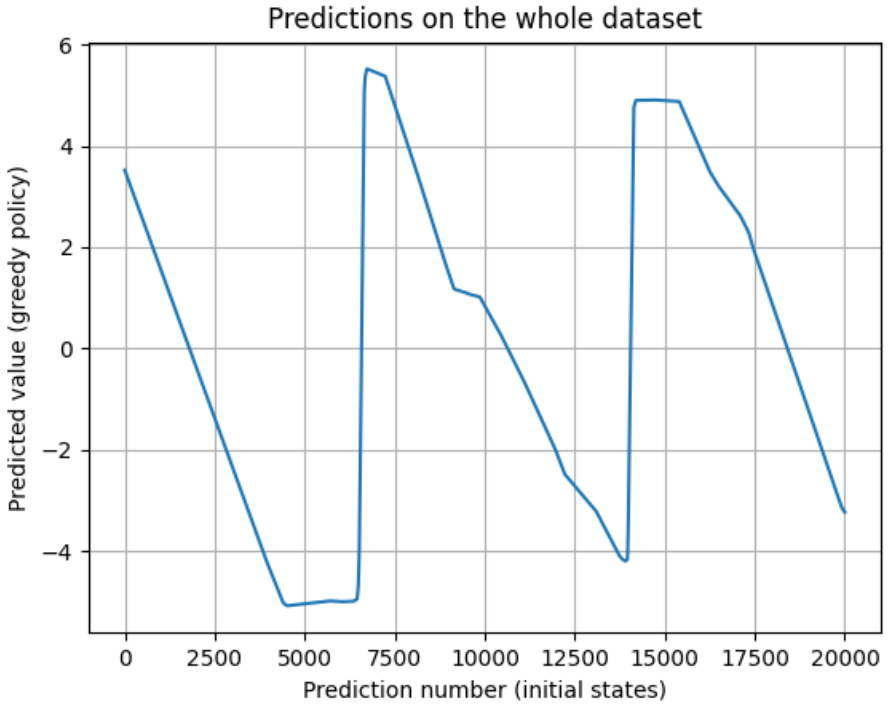


Figure 8: Predictions from the actor network.

The whole training process lasts about 42 seconds. The time taken is reasonably small considering the accuracy reached after training.

4 Testing actor performance

This section aims at measuring the performance of the actor comparing it with the first batch of OCPs. Two different methods have been followed to test actor's performance:

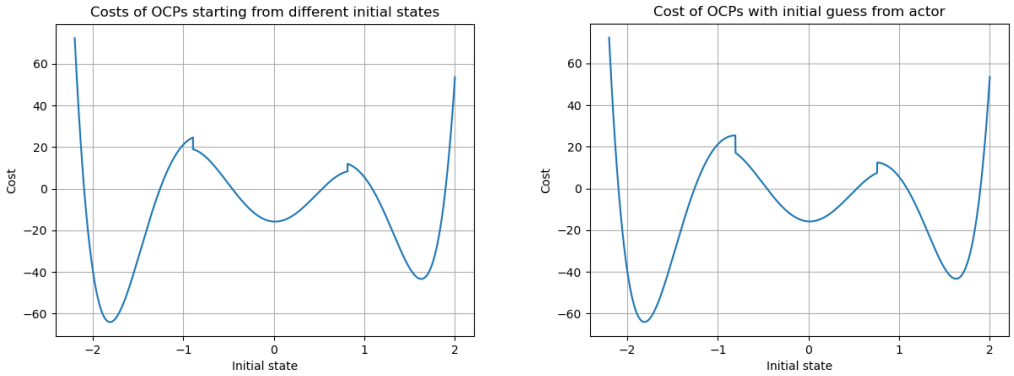
1. Actor's predictions suggested as initial guess to the OCP solver;
2. Actor's predictions used to directly control the system.

In this case, a smaller batch of problems has been considered (2000 instead of 20000) because of the computational time requested by the algorithm. The selected number is a reasonable trade-off between accuracy of the results (especially visually comparing the plots) and requested time. In particular, the process takes about 20 minutes to be completed, which is fairly acceptable.

4.1 Actor's predictions as initial guess

For each initial state, a set of states and controls have been computed from time instant 0 to N (horizon size) and saved into matrices (one for the states and one for the controls) in which each row corresponds to a different initial state and each

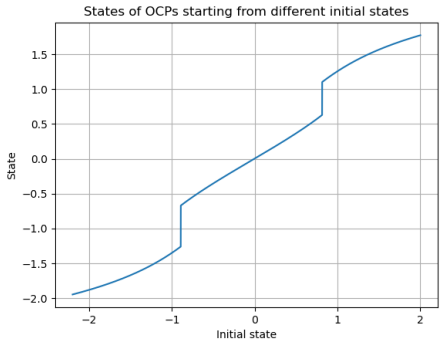
column is related to a specific time instant. At the current time instant, the actor predicts the control to be applied and the next state can be computed starting from the current one and using the predicted control. The cost is updated summing the running cost at each iteration to the previous cost value. The algorithm has been partially optimized using the actor predictions provided by the *.npz* file saved at the end of the actor training process, avoiding to predict again the controls for the first time instant.
At the end of the process, the results (both in states and controls) have been suggested as initial guess to the OCP solver. Figure 9 compares the final cost of the OCPs with or without initial guess.



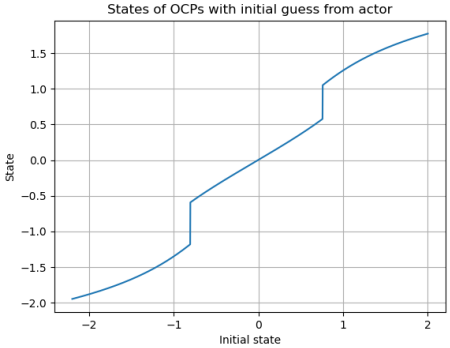
(a) Cost without initial guess. (b) Cost with initial guess.

Figure 9: OCPs cost comparison.

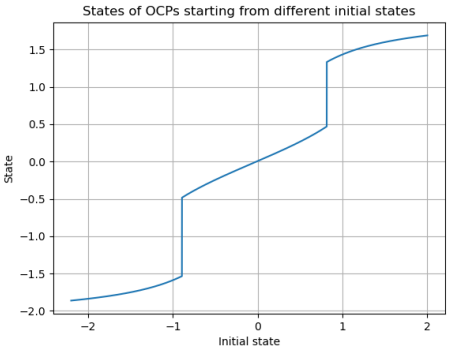
The cost computed exploiting the initial guess is slightly different from the original one, especially in the central portion where some states close to the original local maximum are shifted towards the lateral minimum. The central portion is narrower with respect to the one without initial guess. This means that, even if not so evidently, the actor helps the solver to seek for a global minimum cost (or a more convenient one) thanks to the minimization of the action-value function.
In Figure 10 and 11 some states and controls related to the same time instant in the two cases are compared.



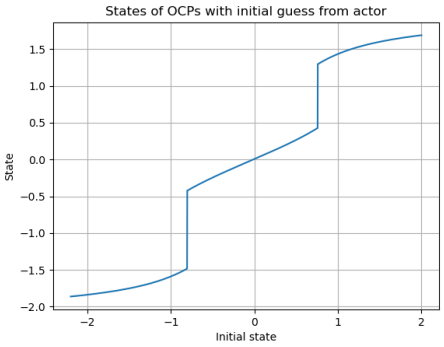
(a) First time step.



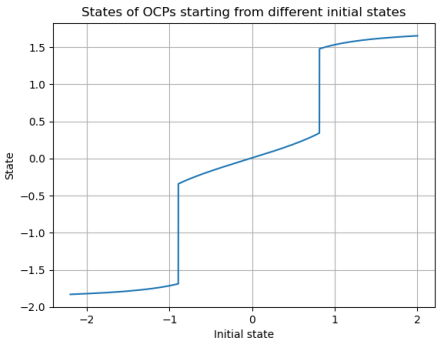
(b) First time step.



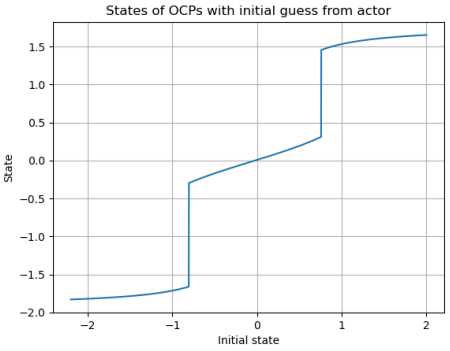
(c) Second time step.



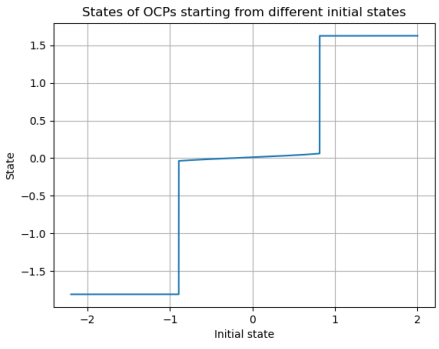
(d) Second time step.



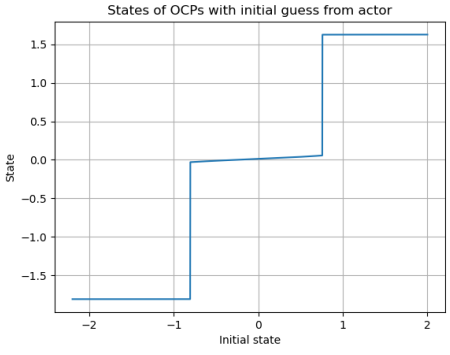
(e) Third time step.



(f) Third time step.

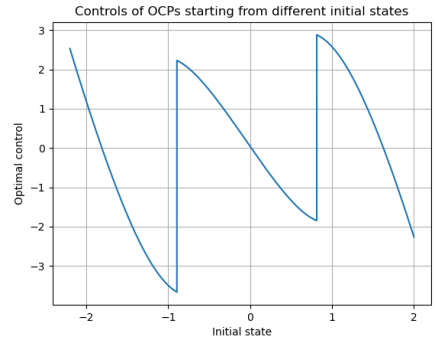


(g) Last time step.

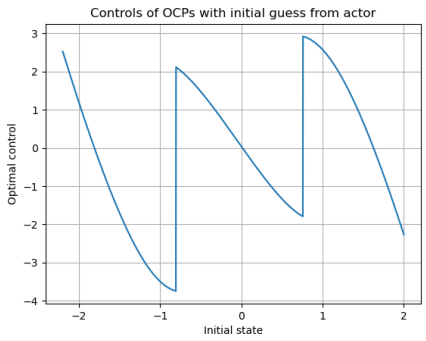


(h) Last time step.

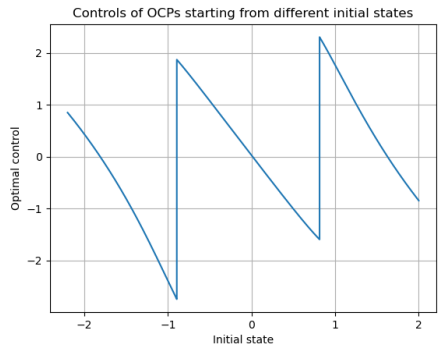
Figure 10: Comparison between states without initial guess (left) and with initial guess (right).



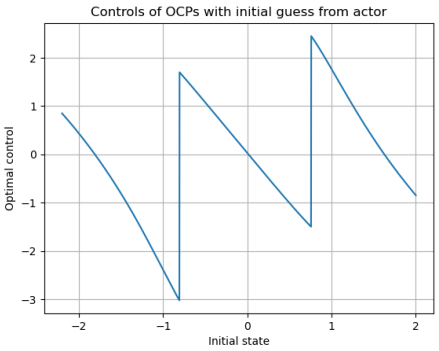
(a) First time step.



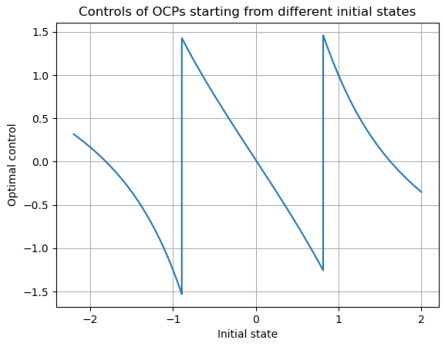
(b) First time step.



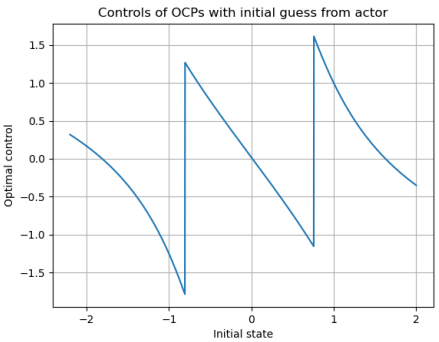
(c) Second time step.



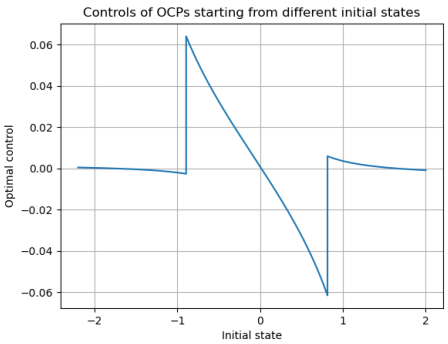
(d) Second time step.



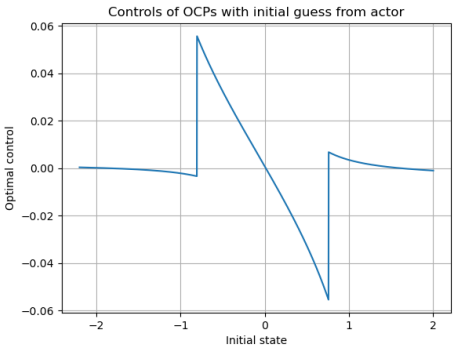
(e) Third time step.



(f) Third time step.



(g) Last time step.

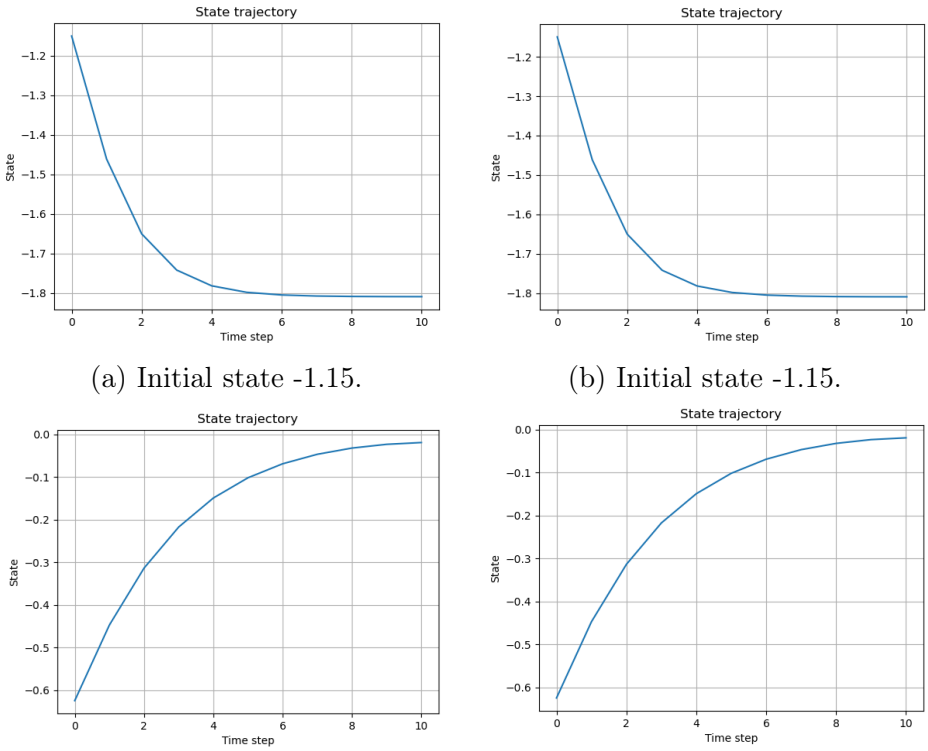


(h) Last time step.

Figure 11: Comparison between controls without initial guess (left) and with initial guess (right).

The differences between states and controls with or without initial guess are not so

evident, but it can be noticed that the initial guess suggests to the solver to apply a slightly larger control in the first time instants to make the states converge faster and reduce the action-value function. This is way more evident in the next Section, where the actor's predictions are applied directly to control the system. Moreover, the set of states which move towards the central position is narrower.
Figure 12 compares two trajectories starting from the same initial states computed with or without the initial guess. In the considered cases, there are no significant differences, but it may depend on the initial state to which they are related.



(a) Initial state -1.15.

(b) Initial state -1.15.

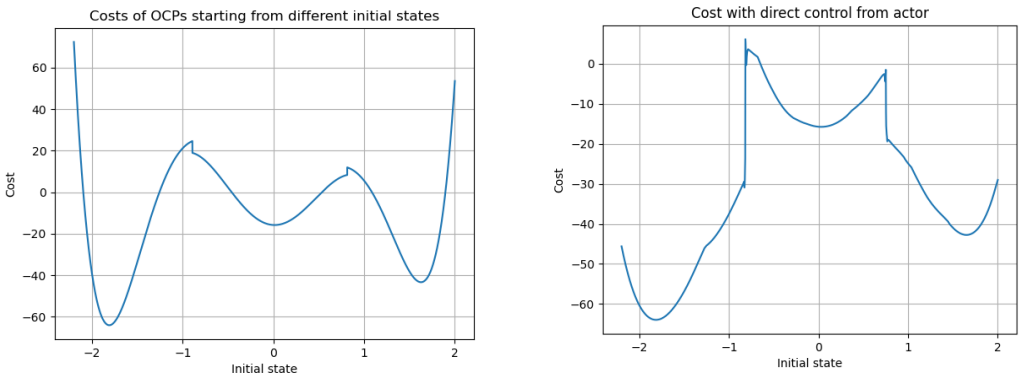
(c) Initial state -0.625.

(d) Initial state -0.625.

Figure 12: Comparison between state trajectories without initial guess (left) and with initial guess (right).

4.2 Actor's predictions as direct control

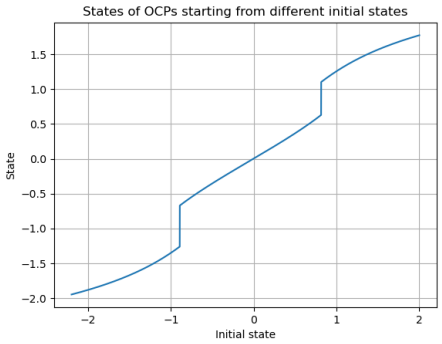
The same algorithm explained in the previous Section has been used to compute states and controls based on actor's predictions and directly control the system.
Figure 13 compares the final cost of the original batch of OCPs and of the system directly controlled by the actor.



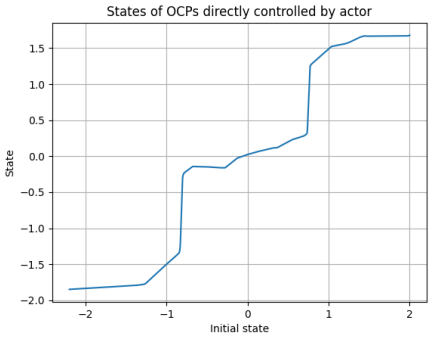
(a) Original OCPs cost. (b) Direct control cost.

Figure 13: Cost comparison.

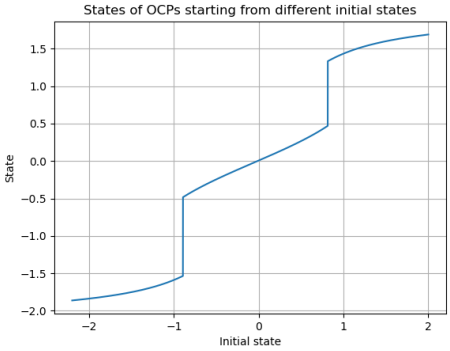
In this case, the differences between the two results are more evident. The cost resulting from applying directly actor's prediction as control has a less smooth shape, possibly causing some problems in a practical case. On the other hand, the cost is way smaller in a large set of cases, with a remarkable difference especially around the global minimum (on the left). The central portion still remains, but it is slightly narrower. The cost of the problems starting from initial states close to the local maximum rapidly drops towards the minimum.
Figure 14 and 15 compare some states and controls related to the same time instant in the two cases.



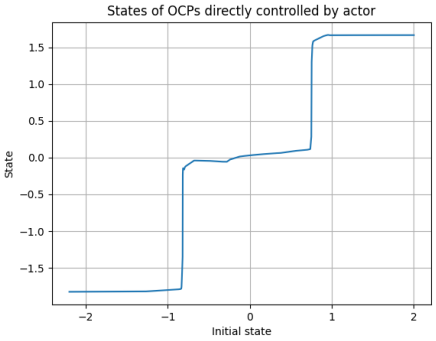
(a) First time step.



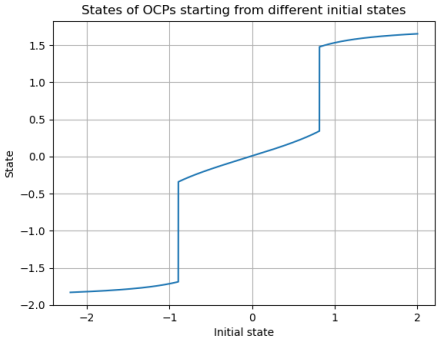
(b) First time step.



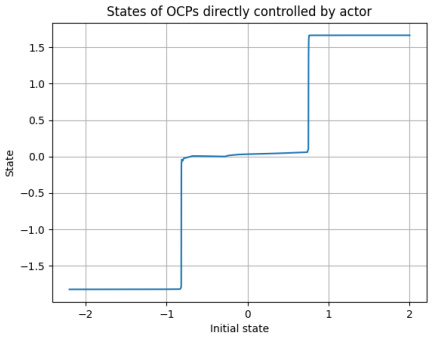
(c) Second time step.



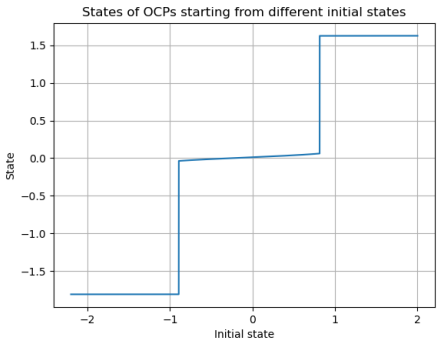
(d) Second time step.



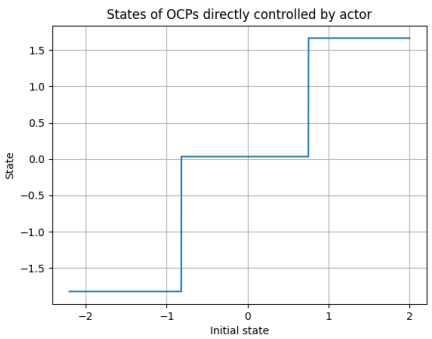
(e) Third time step.



(f) Third time step.

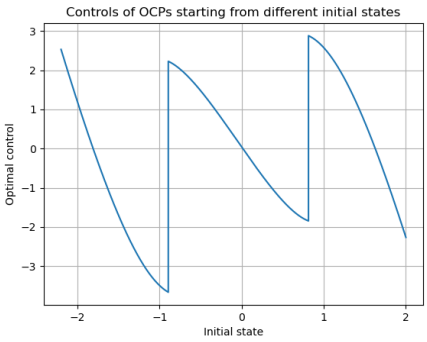


(g) Last time step.

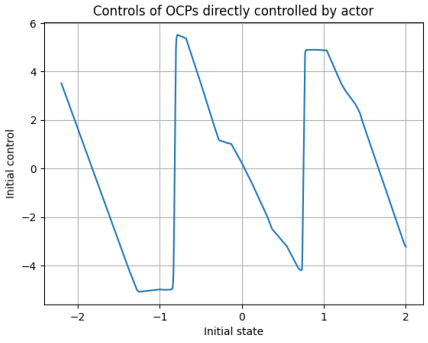


(h) Last time step.

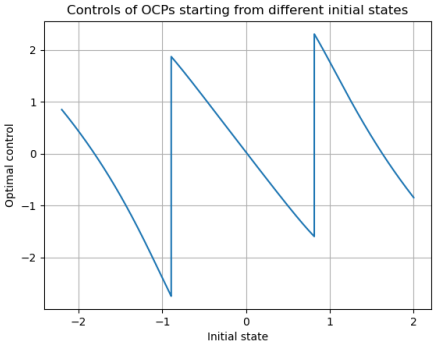
Figure 14: Comparison between states in the original set of OCPs (left) and with direct control from actor's predictions (right).



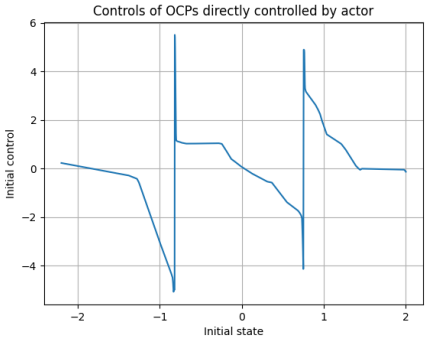
(a) First time step.



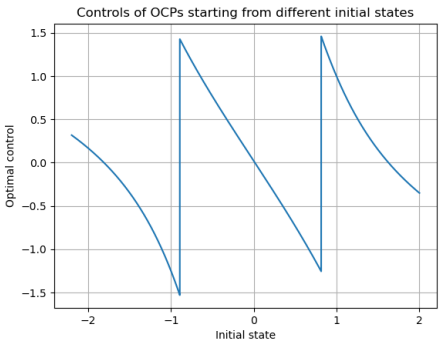
(b) First time step.



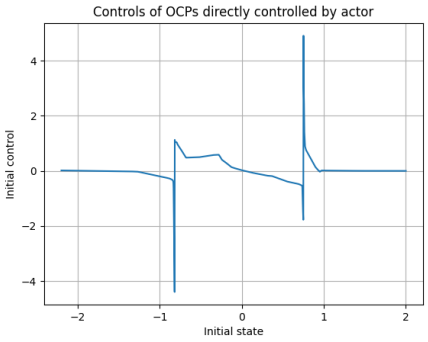
(c) Second time step.



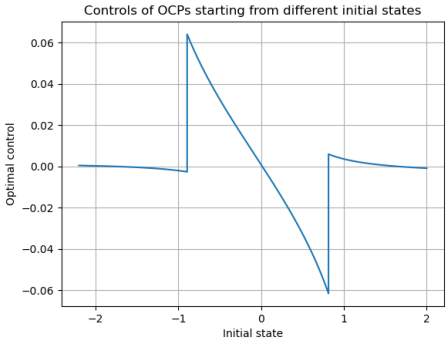
(d) Second time step.



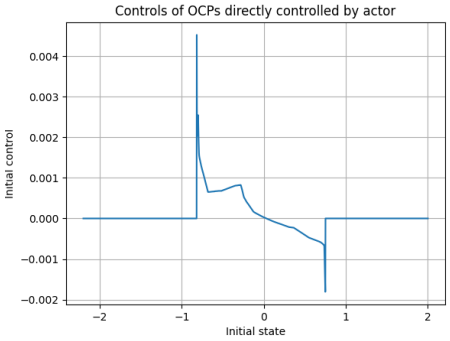
(e) Third time step.



(f) Third time step.



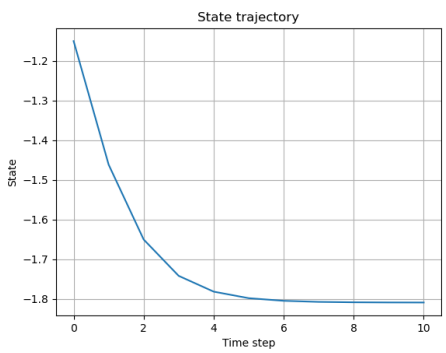
(g) Last time step.



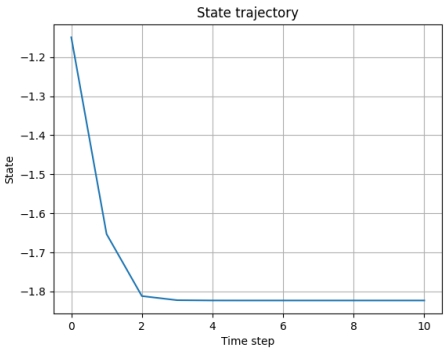
(h) Last time step.

Figure 15: Comparison between controls in the original set of OCPs (left) and with direct control from actor's predictions (right).

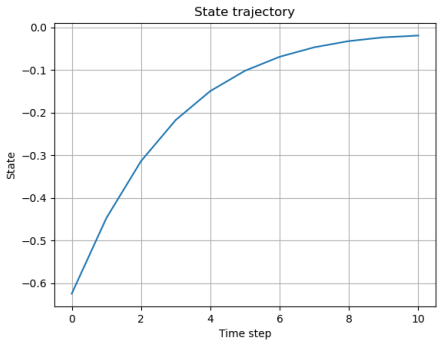
It can be noticed that the states converge way faster applying directly the actor's predictions as control, but there may be some issues related to the discontinuities of the results. In particular, for some specific cases the requested control to be applied may be larger than needed. Despite that, in general the requested controls are initially higher and then much smaller in the next time steps, minimizing the action-value function. For example, the predicted control in the last time step assumes values which are at least 1 order of magnitude smaller than the controls provided by the OCP solver.
Figure 16 compares two trajectories starting from the same initial states computed by the OCP solver or directly controlled by the actor. States converge much faster in the second case, as already observed in the previous plots.



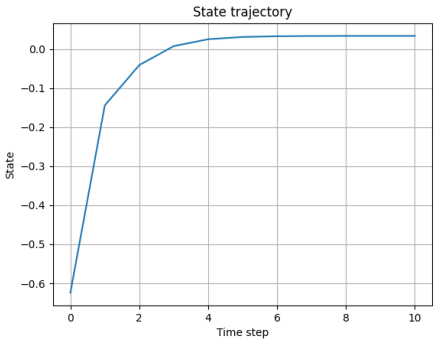
(a) Initial state -1.15.



(b) Initial state -1.15.



(c) Initial state -0.625.



(d) Initial state -0.625.

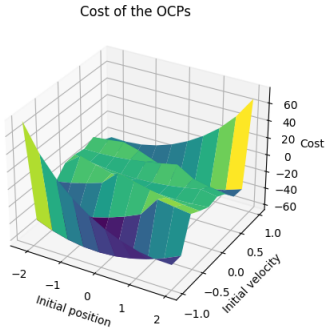
Figure 16: Comparison between state trajectories in the original set of OCPs (left) and with direct control from actor's predictions (right).

5 OCP with double integrator

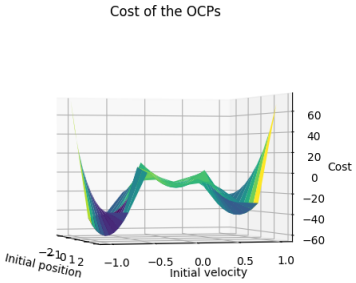
Since the tests made with the single integrator seem to work fine, the next step is to analyse again the system using a double integrator. The dynamics of the system can be modelled as:

$$\begin{bmatrix} p_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_t \\ v_t \end{bmatrix} + \begin{bmatrix} 0.5\Delta t^2 \\ \Delta t \end{bmatrix} u_t$$

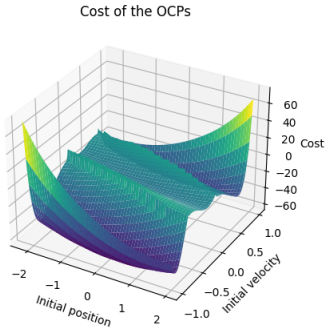
while the running cost is described by the same equation of the single integrator (Equation 2), depending on current position and control and resulting in a non-convex problem. Horizon size has been set to 10, as for the single integrator.
Many OCPs have been solved starting from different initial states ranging from a minimum of -2.2 to a maximum of 2 for the initial position and from -1 to 1 for the initial velocity. The whole range of possible initial states has been divided in a grid, with states equally spaced from each other. Controls can assume values ranging from -5 to 5, as for the single integrator. CasADi has been exploited to solve the OCPs explicitly extracting cost and optimal control of the OCPs for each and every initial state and saving the results into a .npz file. The accuracy of the results depends also on the number of considered initial states. A reasonable trade-off between requested computational time and accuracy can be achieved choosing a 100x100 grid of initial states (100 position and 100 velocity values between the selected boundaries). Moreover, we noticed that enlarging the number of initial states decreases the number of epochs needed to train the neural network (because of the larger number of information provided to the network). Figure 17 compares some cost plots of the solved OCPs using smaller or larger sets of initial states. For a better analysis of what happens in the system with a double integrator, every combination of position and velocity has been taken into account, leading to a 3D plot. The computational time requested to solve the OCPs with a 100x100 initial states grid is less than five minutes, while almost 8 hours were necessary only to compute the OCPs in the 1000x1000 case.



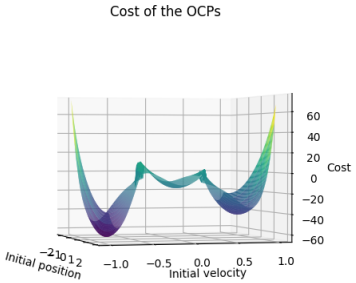
(a) Cost of OCPs on a 10x10 grid.



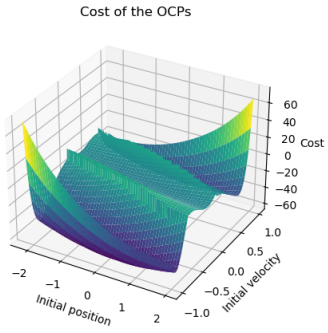
(b) Cost of OCPs on a 10x10 grid.



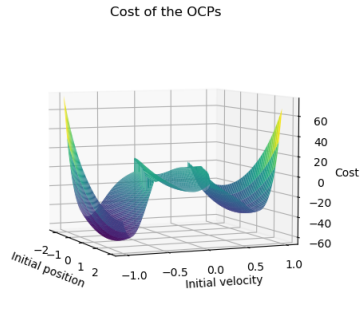
(c) Cost of OCPs on a 100x100 grid.



(d) Cost of OCPs on a 100x100 grid.



(e) Cost of OCPs on a 1000x1000 grid.

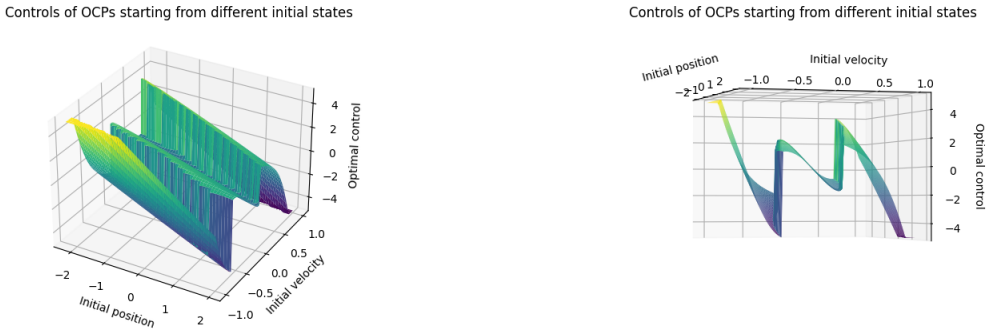


(f) Cost of OCPs on a 1000x1000 grid.

Figure 17: Costs of OCPs starting from different initial states sets.

It can be observed that there are some differences in cost results between using a single integrator or a double integrator. This is due to the fact that position and velocity are coupled, meaning changes in velocity affect position and vice versa, which can introduce complexities in the control strategies and affect the cost optimization compared to the dynamics of a single integrator. Moreover, the constraints imposed on a double integrator involve both position and velocity limits and, for example, constraints on velocity might limit the achievable optimal cost compared to constraints on position alone. In particular, increasing velocity bounds from (-1,1) to (-5,5), the cost significantly increases because of the limited capability of controlling the system if velocity is high. Lastly, also the time taken for the computation of the

results with a double integrator is higher than a single integrator, due to a higher complexity.
Plotting the optimal control computed by the OCPs solver, we get the behaviour in Figure 18.

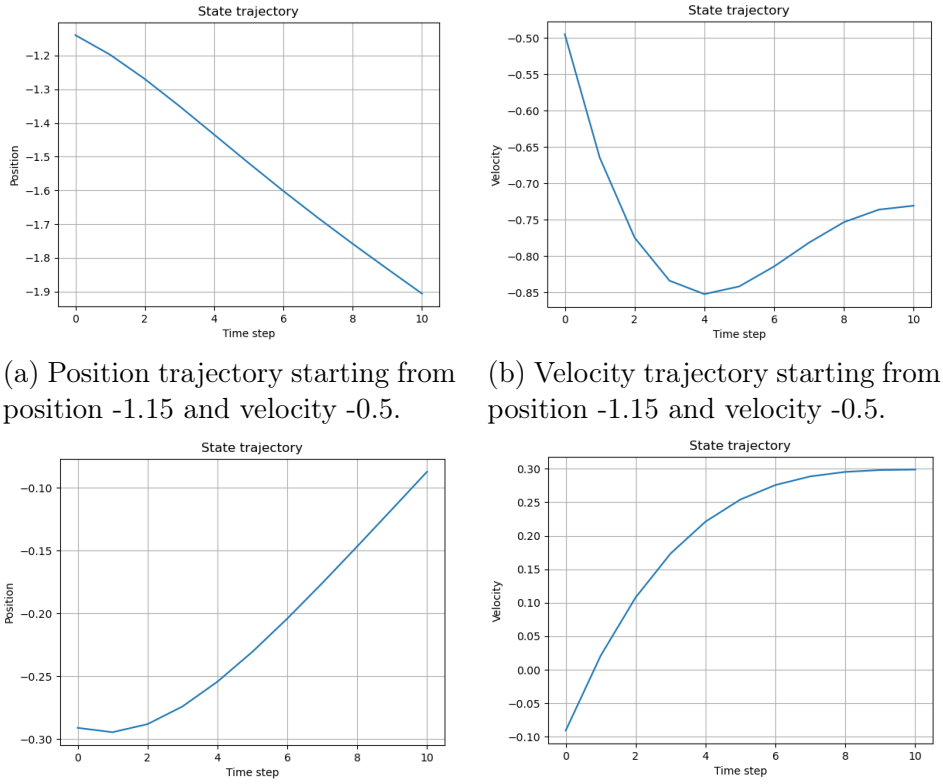


(a) Top view.

(b) Front view.

Figure 18: Optimal control of OCPs starting from different initial states.

Figure 19 shows two examples of state trajectories computed by the OCP solver.



(a) Position trajectory starting from position -1.15 and velocity -0.5.

(b) Velocity trajectory starting from position -1.15 and velocity -0.5.

(c) Position trajectory starting from position -0.29 and velocity -0.1.

(d) Velocity trajectory starting from position -0.29 and velocity -0.1.

Figure 19: State trajectory examples.

6 Neural network "critic" in double integrator case

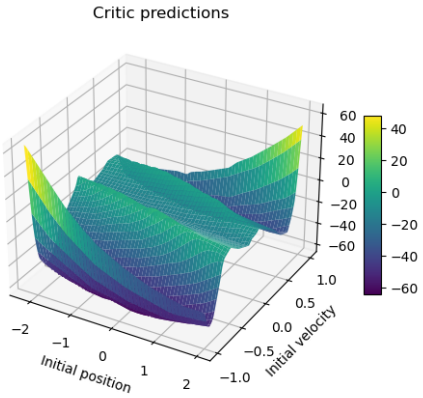
The aim of the critic is the same described in Section 2. The structure of the critic network is the following:

- Input layer: composed of 2 neurons, one that considers the positions and the other that considers the velocities. These two inputs are flattened before being sent to the first hidden layer.
- Hidden layer 1, with 16 neurons
- Hidden layer 2, with 64 neurons
- Hidden layer 3, with 32 neurons
- Output layer: 1 neuron which stores the predicted cost.

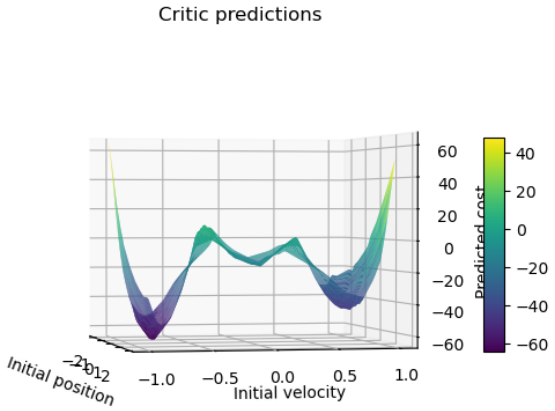
With multi-input neurons in a neural network it is generally suggested to start with a higher number of neurons and decrease the number until the output is reached in order to abstract and summarize the learned features while moving deeper into the network. However, in this case it has been observed that a network with lower number of neurons in the first hidden layer and a higher number of neurons in the second and third layers performs better. No benefits have been achieved trying different configurations, since this solution is the best compromise between complexity of the network structure and accuracy of results, paying attention at avoiding overfitting of data.
Regarding the training of the critic network with a double integrator, the batch size chosen is 32, while the selected number of epochs is 100. Also in this case we let the built-in functions of Tensorflow to estimate the optimal learning rate. As for the single integrator, the activation function chosen for each layer is the ReLu (Rectified Linear Activation Unit), while for the loss function the mean squared error has been chosen.

6.1 Results

The predicted values (Figure 20) follow a behaviour similar to the true one, with some small differences. This is due to the fact that the neural network still has some loss, which can be further limited with optimization of the number of layers, of the number of neurons and of the hyperparameters. To do so, it is suggested to use a tool called "Optuna", which is able to automatically tune all the desired hyperparameters in order to get the minimum loss and the best results in training.



(a) Cost predicted from critic (top view).



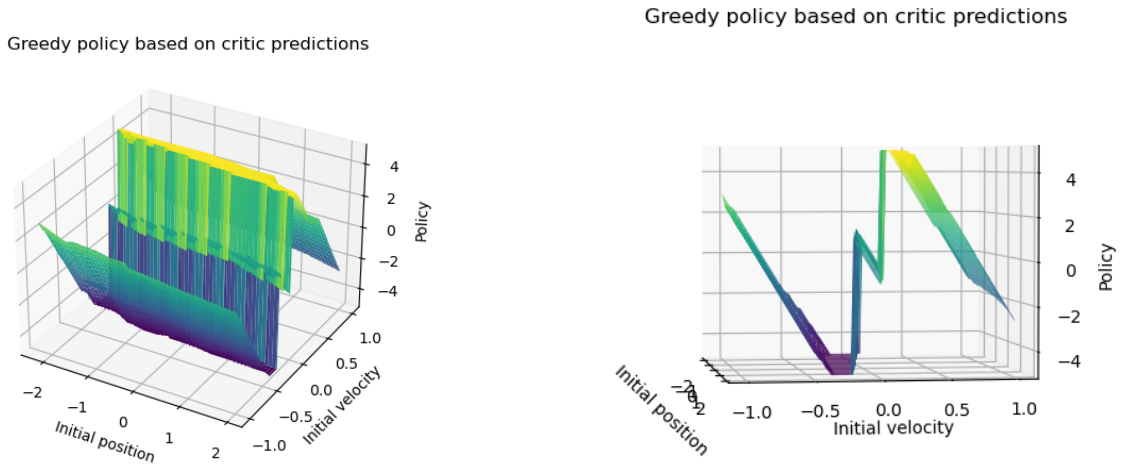
(b) Cost predicted from critic (front view).

Figure 20: Cost predicted from critic network.

The time taken for training and testing the network is 71 seconds for the 100x100 grid. In the 1000x1000 case, the training process lasts more than 14 minutes with 20 epochs.

7 Neural network "actor" in double integrator case

Once the critic has been trained as in Section 6, the actor should be trained to approximate the greedy policy with respect to the critic.
As already explained in Section 3, the action-value function corresponding to the critic should be minimized in order to find the policy π following Equation 3. For each initial state, a possible state at the next time step can be computed through the double integrator equation. The greedy policy with respect to the critic has been found searching for the minimum value of the action value function depending on the control. Discount factor has been taken as 1 to focus more on final cost rather than immediate cost. This phase takes 15 minutes, being the slowest part of the process. An array containing all the computed values for $\pi(x)$ has been saved in a *.npz* file and the results are shown in Figure 21.



(a) Greedy policy (top view).

(b) Greedy policy (front view).

Figure 21: Greedy policy with respect to the critic.

The actor's neural network is built as follows, taking into account the complexity of the problem and the accuracy of results:

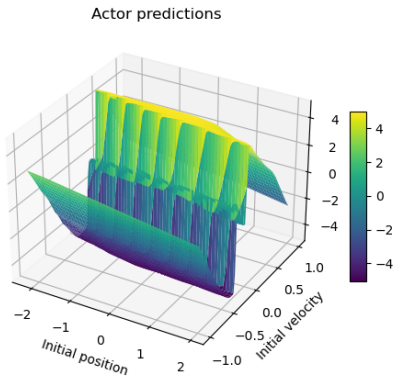
- Input layer: composed of 2 neurons (position and velocity), which are flattened before being sent to the first hidden layer.
- Hidden layer 1, with 16 neurons
- Hidden layer 2, with 64 neurons
- Hidden layer 3, with 32 neurons
- Output layer: 1 neuron which stores the approximation of the greedy policy.

Also in this case some other alternatives have been taken into account, but the best is the one described above, which is the same as the critic's network, giving the best compromise between prediction loss and computation time.
Activation and loss functions are the same as the previous built neural networks. The hyperparameters have been tuned as follows:

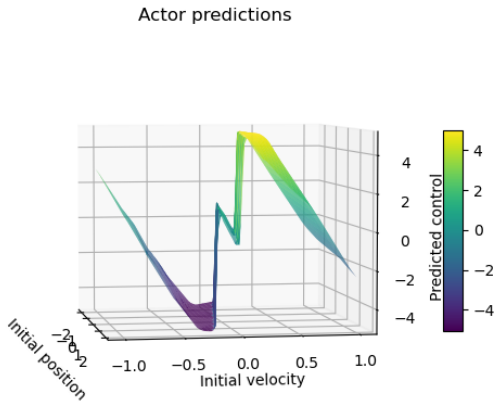
- Batch size = 32
- Epochs = 100
- Learning rate = automatically chosen by the built-in functions in Tensorflow.

7.1 Results

The test phase in an unseen set of data has reported good results and a minimum loss, hence we made a prediction on the whole dataset, which is reported in Figure 22.



(a) Predictions from actor (top view).



(b) Predictions from actor (front view).

Figure 22: Actor's predictions on the whole dataset.

The predicted values have been stored in a *.npz* file. The plot is slightly different from Figure 21 because of the actor's predictions.
The whole training process lasts 64 seconds. The time taken is reasonably small considering the accuracy reached after training.

8 Actor performances in double integrator case

This section aims at measuring the performance of the actor comparing it with the first batch of OCPs. Also in this case, two different methods have been followed to test actor's performance:

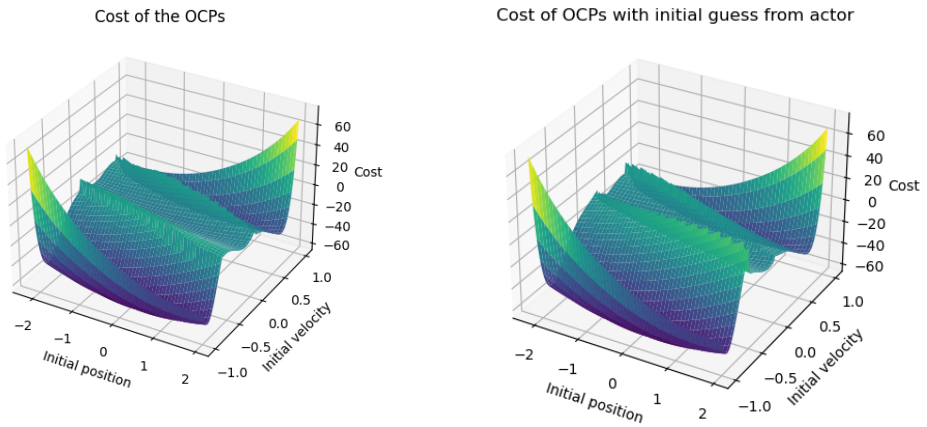
1. Actor's predictions suggested as initial guess to the OCP solver;
2. Actor's predictions used to directly control the system.

The algorithm has been optimized in order to minimize the computational time, calling the "model.predict()" function as few times as possible (making predictions on the whole set of states instead of one state at a time). In this way, the requested time for computing 10000 trajectories directly controlled by the actor suggestions is less than one minute.

8.1 Actor's predictions as initial guess

As for the single integrator, for each initial state a set of states and controls have been computed from time instant 0 to N (horizon size) and saved into matrices. For states, two 3-dimensional matrices have been created (one for positions and one for velocities): the first two dimensions are related to the initial state (initial position + initial velocity) while the third one is related to the time step (from 0 to N). For controls, a 3-dimensional matrix has been created too, with the third dimension related to each time step (from 0 to N-1) and containing the scalar value of the control. The cost is updated summing the running cost at each iteration to the previous cost value.

At the end of the process, the results (both in states and controls) have been suggested as initial guess to the OCP solver. The solver takes 8 minutes (instead of 10) to solve the problems helped by actor's initial guess. Figure 23 compares the final cost of the OCPs with or without initial guess.

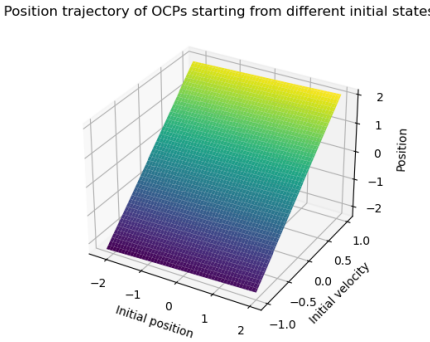


(a) Cost without initial guess.

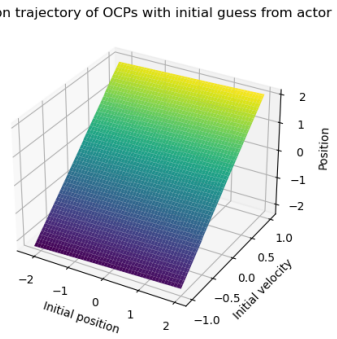
(b) Cost with initial guess.

Figure 23: OCPs cost comparison.

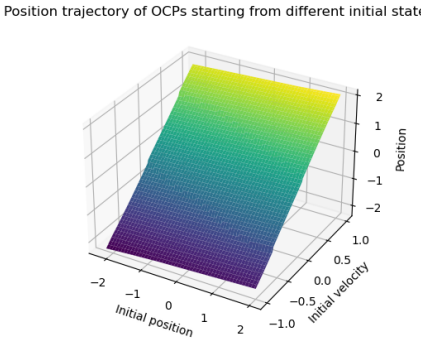
As already observed in the single integrator case, the cost computed exploiting the initial guess is slightly different from the original one, especially in the central portion which is narrower. The minimization of the action-value function helps the solver to seek for a global minimum cost (or a more convenient minimum).
In Figure 24, 25 and 26 some states (position and velocity) and controls related to the same time instant in the two cases are compared.



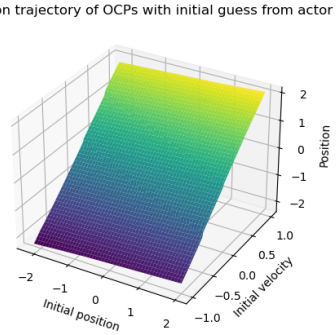
(a) First time step.



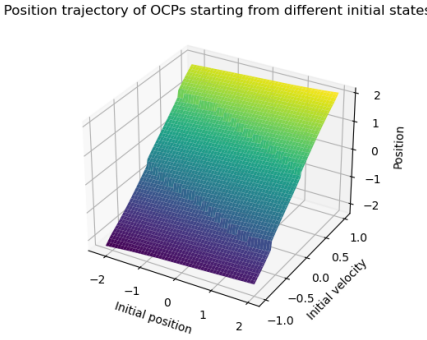
(b) First time step.



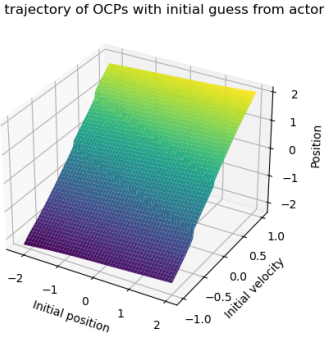
(c) Second time step.



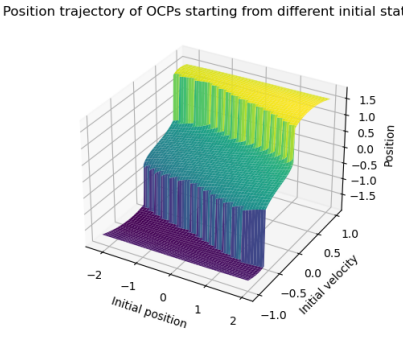
(d) Second time step.



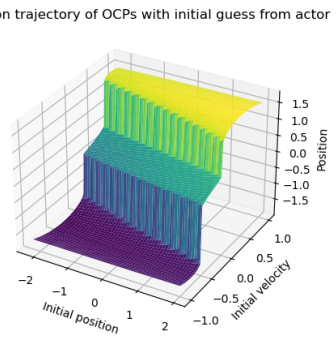
(e) Third time step.



(f) Third time step.

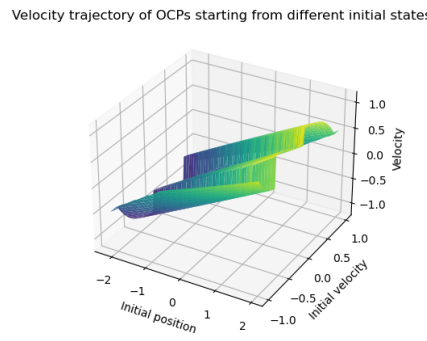


(g) Last time step.

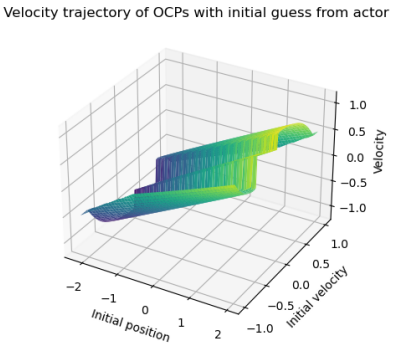


(h) Last time step.

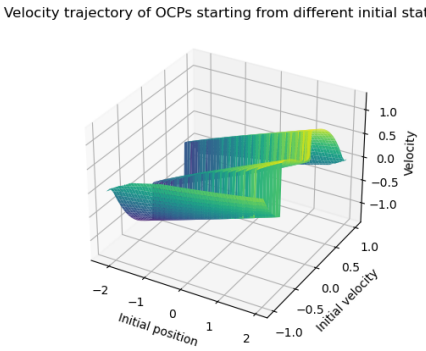
Figure 24: Comparison between states position without initial guess (left) and with initial guess (right).



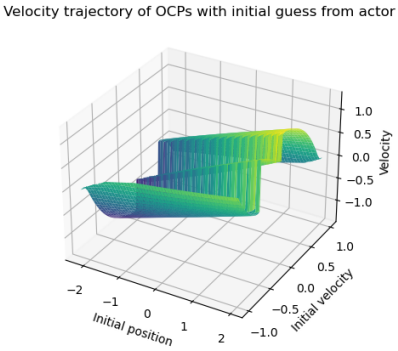
(a) First time step.



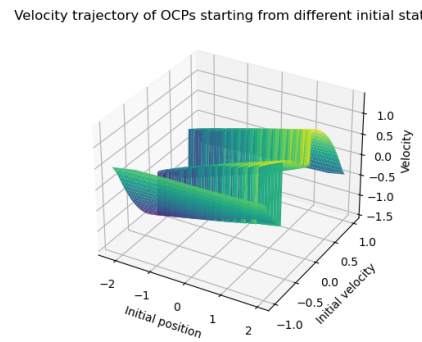
(b) First time step.



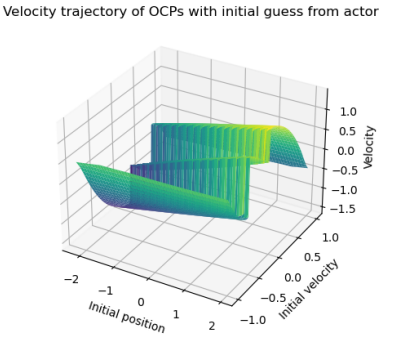
(c) Second time step.



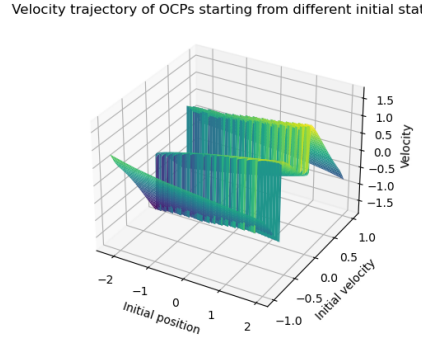
(d) Second time step.



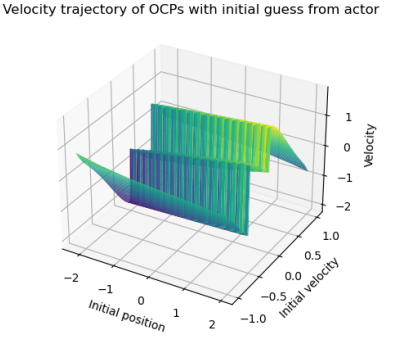
(e) Third time step.



(f) Third time step.

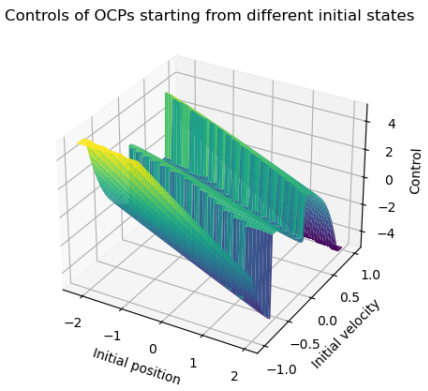


(g) Last time step.

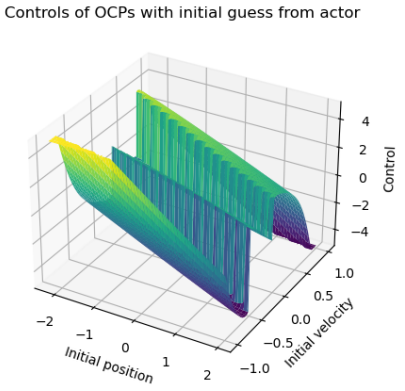


(h) Last time step.

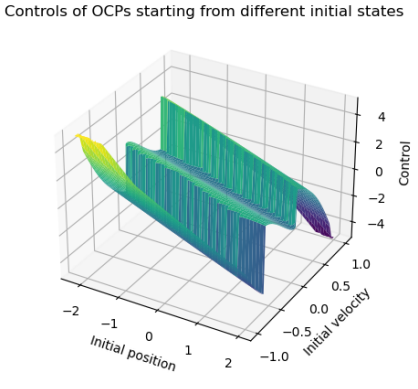
Figure 25: Comparison between states velocity without initial guess (left) and with initial guess (right).



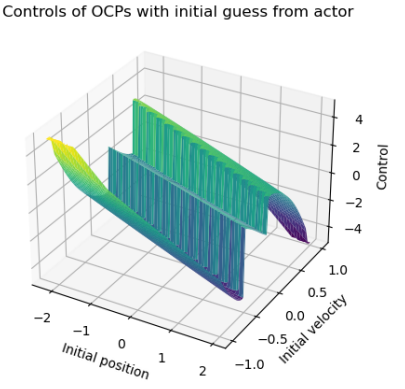
(a) First time step.



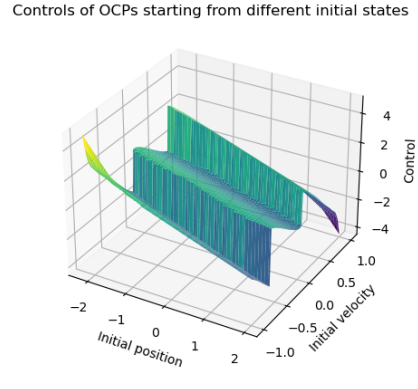
(b) First time step.



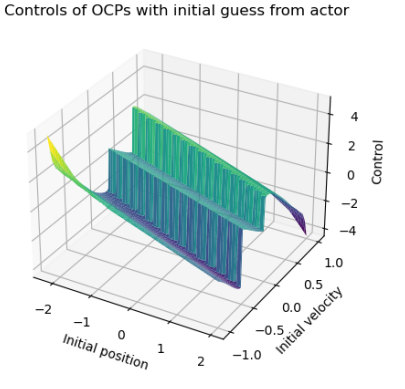
(c) Second time step.



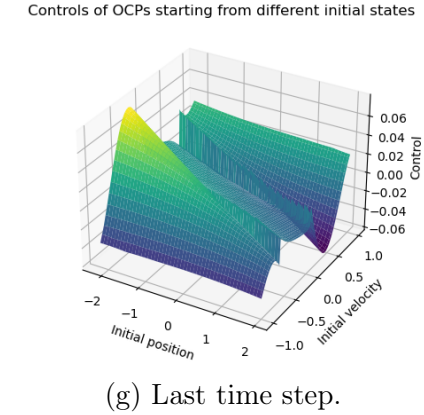
(d) Second time step.



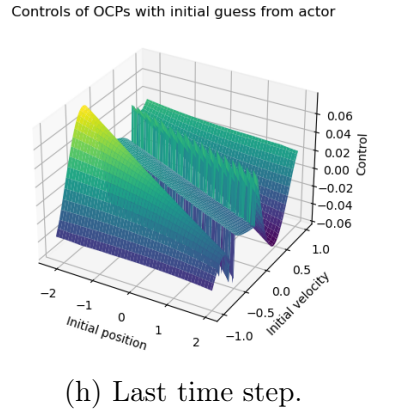
(e) Third time step.



(f) Third time step.



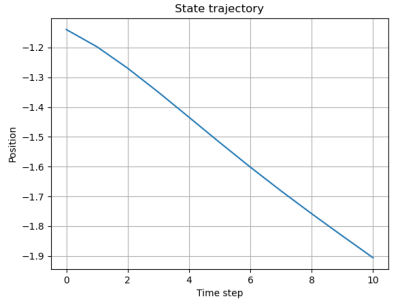
(g) Last time step.



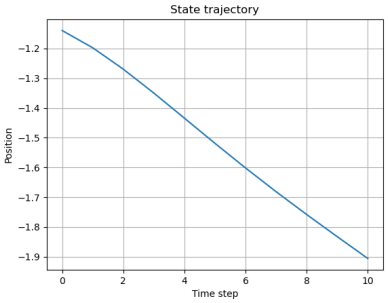
(h) Last time step.

Figure 26: Comparison between controls without initial guess (left) and with initial guess (right).

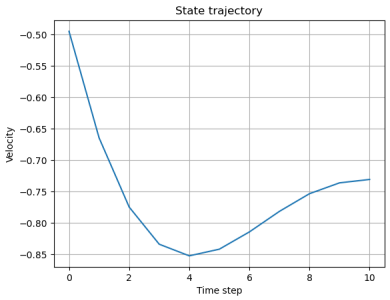
The differences between states and controls with or without initial guess are not so evident, but it can be noticed that the initial guess suggests to the solver to apply a slightly larger control in the first time instants to make the states converge faster and reduce the action-value function. The results at the last time step are a bit different especially regarding position and velocity, tending to be closer to the actor suggestions as it can be observed in the next Section. In particular, position tends to avoid more the central positions and move towards the lateral regions, pointing in the direction of more convenient minimum points.
Figure 32 compares two trajectories starting from the same initial states computed with or without the initial guess. In the considered cases there are no significant differences, but it may depend on the initial state to which they are related.



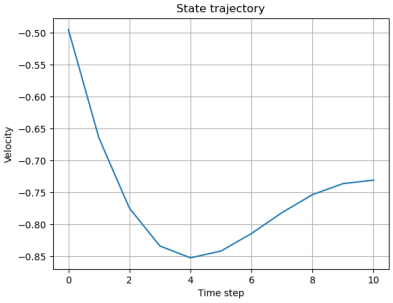
(a) Position trajectory starting from position -1.15 and velocity -0.5.



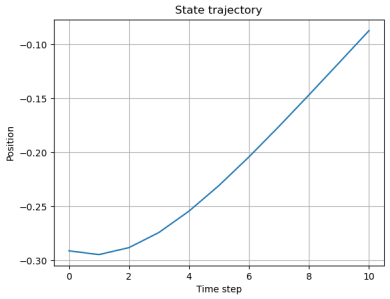
(b) Position trajectory starting from position -1.15 and velocity -0.5 with initial guess.



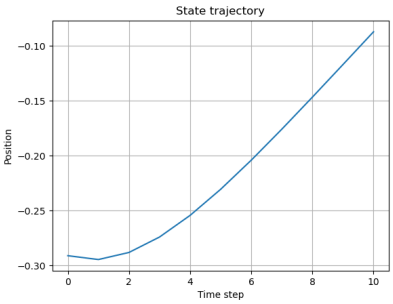
(c) Velocity trajectory starting from position -1.15 and velocity -0.5.



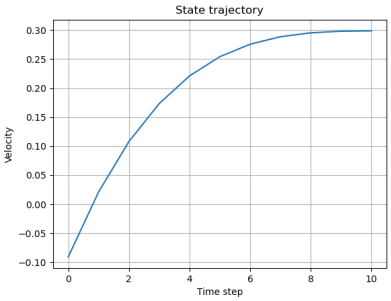
(d) Velocity trajectory starting from position -1.15 and velocity -0.5 with initial guess.



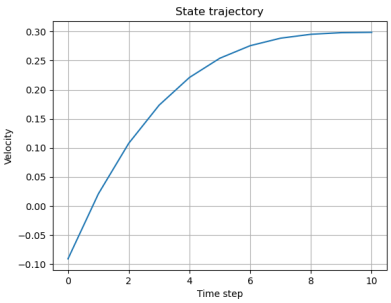
(e) Position trajectory starting from position -0.29 and velocity -0.1.



(f) Position trajectory starting from position -0.29 and velocity -0.1 with initial guess.



(g) Velocity trajectory starting from position -0.29 and velocity -0.1.

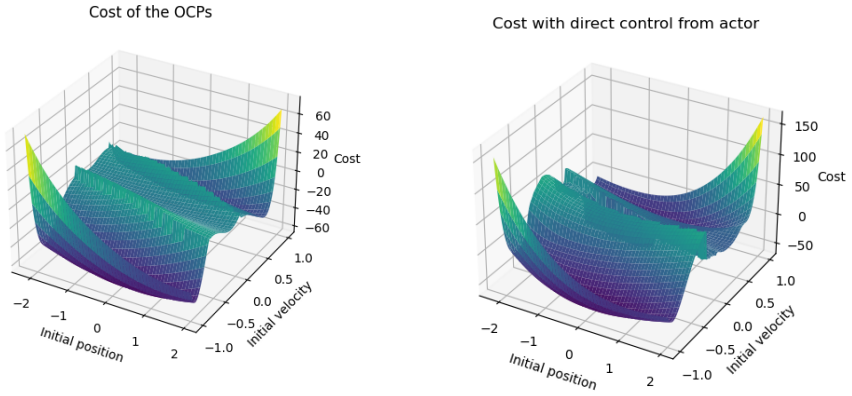


(h) Velocity trajectory starting from position -0.29 and velocity -0.1 with initial guess.

Figure 27: State trajectory comparison without (left) or with (right) initial guess from actor.

8.2 Actor's predictions as direct control

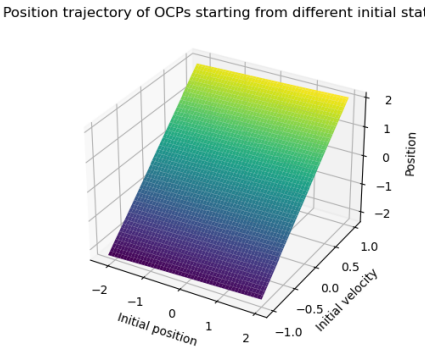
The same algorithm has been used to compute states and controls based on actor's predictions and directly control the system.
Figure 28 compares the final cost of the original batch of OCPs and of the system directly controlled by the actor.



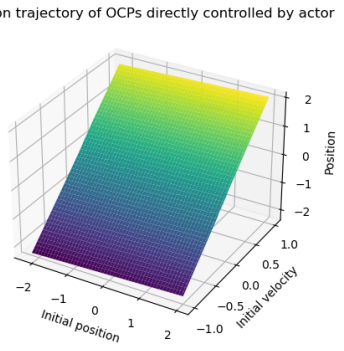
(a) Original OCPs cost. (b) Direct control cost.

Figure 28: Cost comparison.

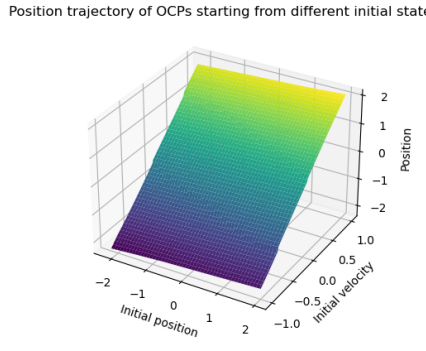
It can be noticed that the cost presents some differences with the original one, reaching much higher values for some states. It seems that the double integrator is not able to minimize the cost starting from the considered set of initial states. The central portion is still narrower with respect to the original one, as happened in the previous Section, meaning that the actor is trying to move towards more convenient minimum points. Despite that, there are evidently some issues with cost minimization. For this reason, it is immediately clear that it may be better to exploit actor's suggestions only as initial guess to warm start the OCP solver instead of directly using them to control the system.
Figure 29, 30 and 31 compare some states (position and velocity) and controls related to the same time instant in the two cases.



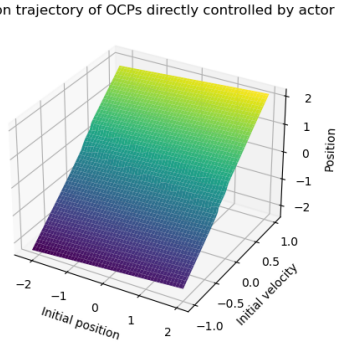
(a) First time step.



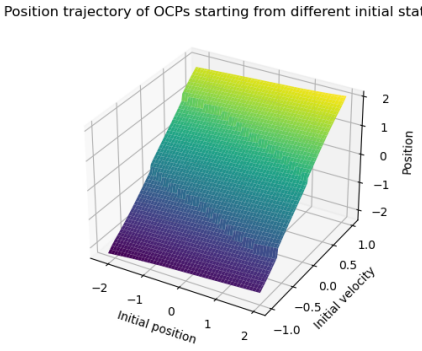
(b) First time step.



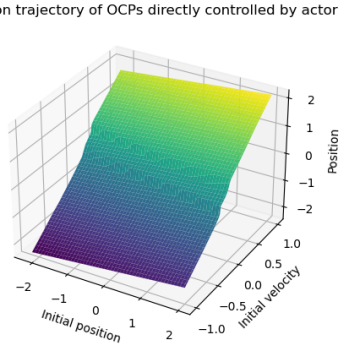
(c) Second time step.



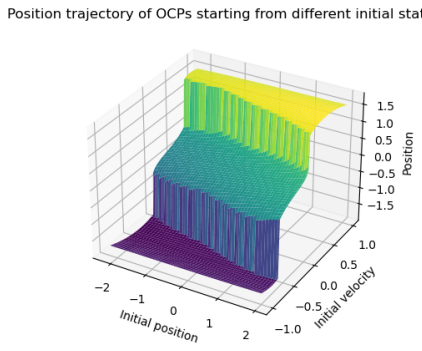
(d) Second time step.



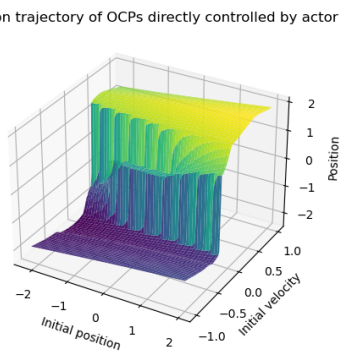
(e) Third time step.



(f) Third time step.

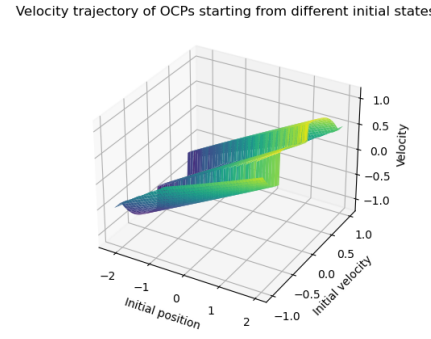


(g) Last time step.

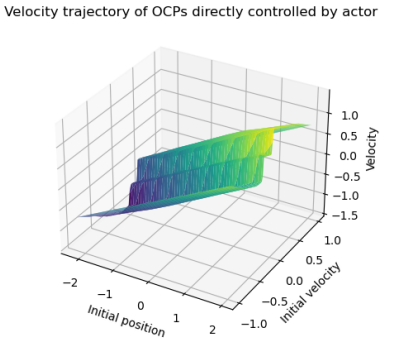


(h) Last time step.

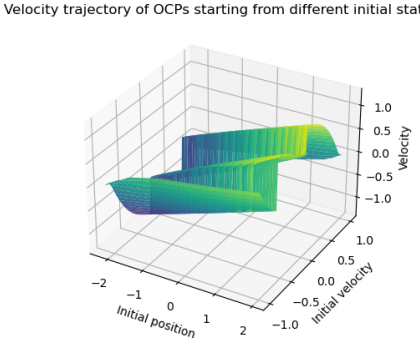
Figure 29: Comparison between state position in the original set of OCPs (left) and with direct control from actor's predictions (right).



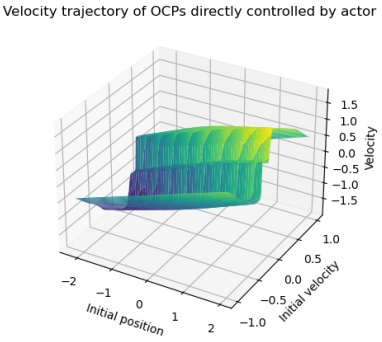
(a) First time step.



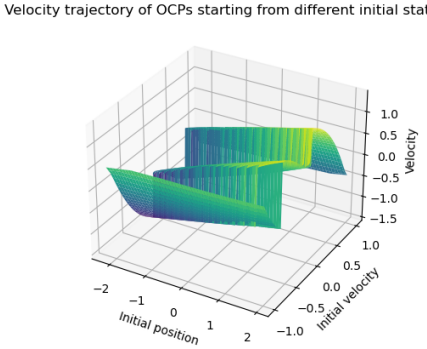
(b) First time step.



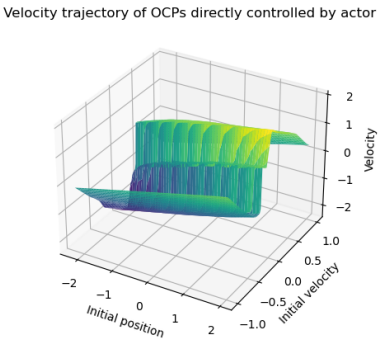
(c) Second time step.



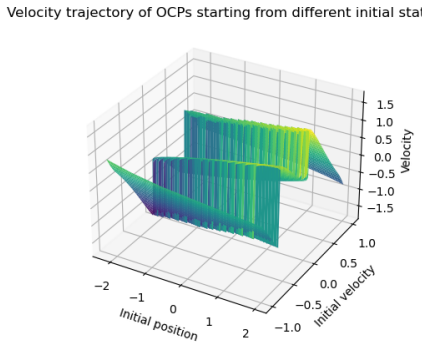
(d) Second time step.



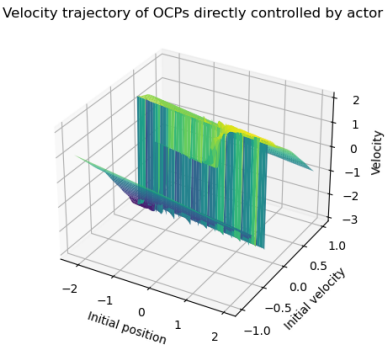
(e) Third time step.



(f) Third time step.

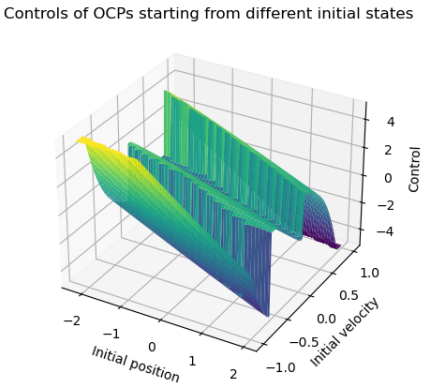


(g) Last time step.

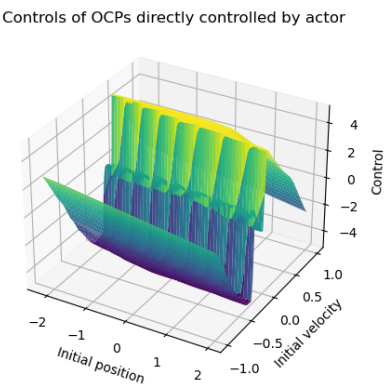


(h) Last time step.

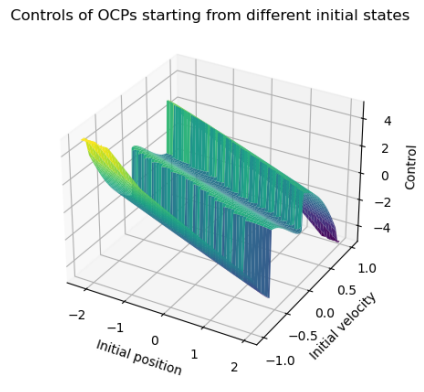
Figure 30: Comparison between state velocity in the original set of OCPs (left) and with direct control from actor's predictions (right).



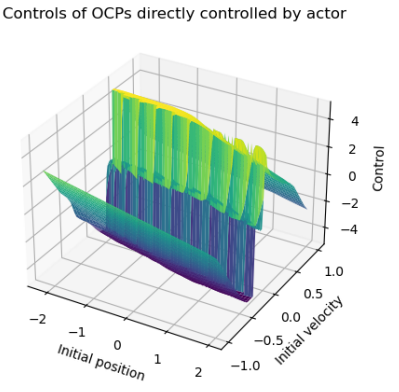
(a) First time step.



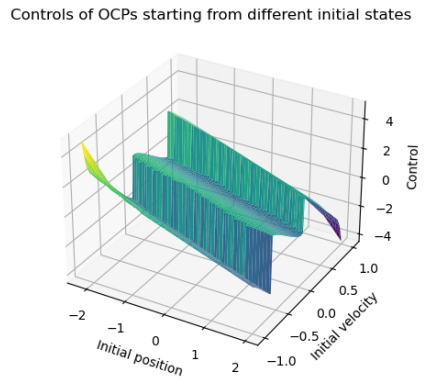
(b) First time step.



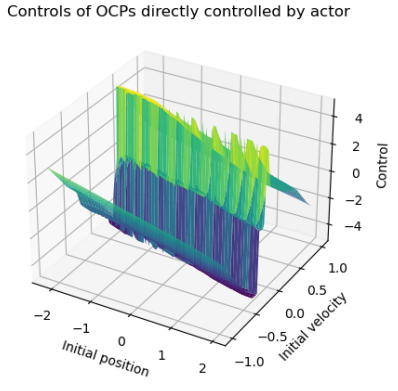
(c) Second time step.



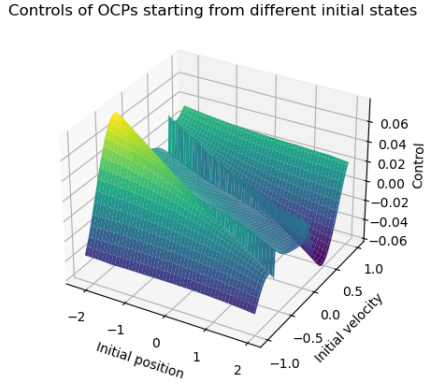
(d) Second time step.



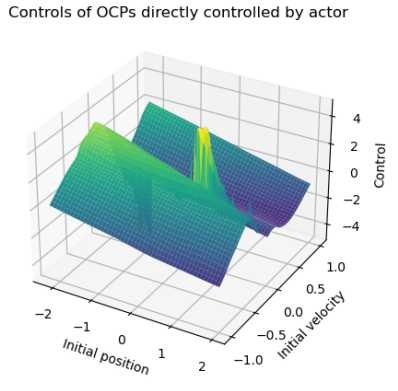
(e) Third time step.



(f) Third time step.



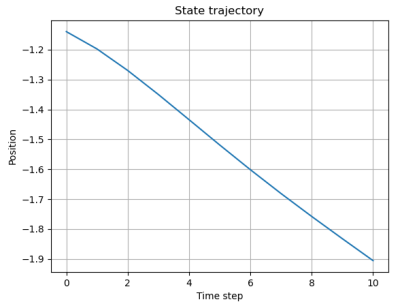
(g) Last time step.



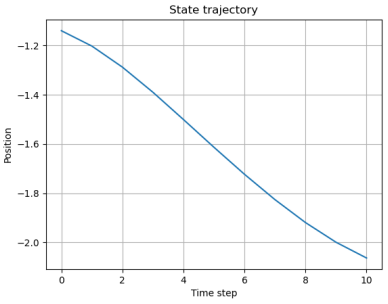
(h) Last time step.

Figure 31: Comparison between controls in the original set of OCPs (left) and with direct control from actor's predictions (right).

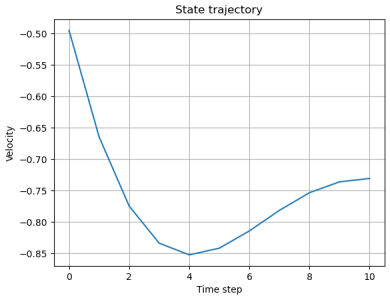
The behaviour of the system directly controlled is really different from the one of the original set of solved OCPs. States don't move towards the central positions as happened in the original OCPs, but tend to shift to the lateral regions. While states remain bounded between -2.2 and 2, velocities increase outside the initial boundaries reaching higher values. Controls don't seem to converge to zero, but stabilize around high values similar to the ones used at the first time instant. This means that the actor is not able to stabilize the system and it may be the reason why also cost is high for many states.
Figure 16 compares two trajectories starting from the same initial states computed by the OCP solver or directly controlled by the actor.



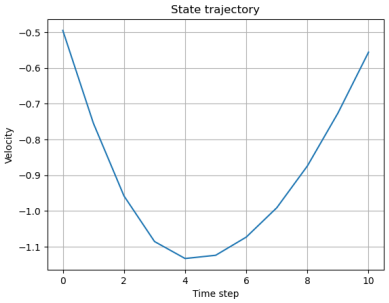
(a) Position trajectory starting from position -1.15 and velocity -0.5.



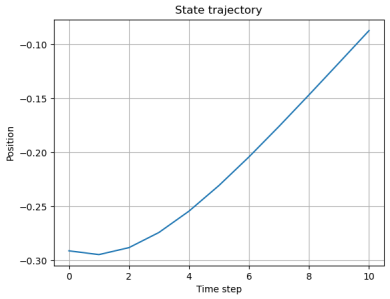
(b) Position trajectory starting from position -1.15 and velocity -0.5 with direct control.



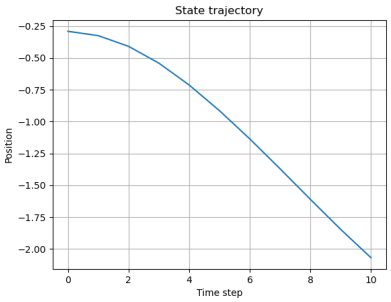
(c) Velocity trajectory starting from position -1.15 and velocity -0.5.



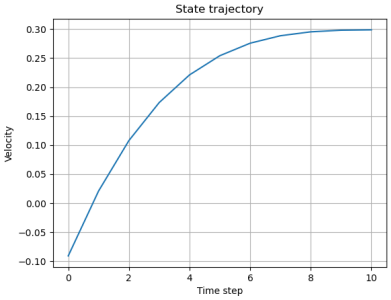
(d) Velocity trajectory starting from position -1.15 and velocity -0.5 with direct control.



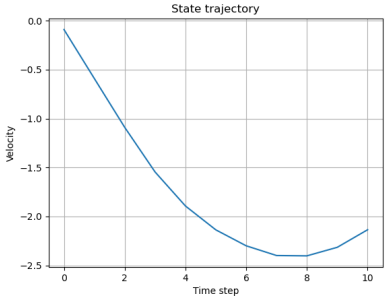
(e) Position trajectory starting from position -0.29 and velocity -0.1.



(f) Position trajectory starting from position -0.29 and velocity -0.1 with direct control.



(g) Velocity trajectory starting from position -0.29 and velocity -0.1.



(h) Velocity trajectory starting from position -0.29 and velocity -0.1 with direct control.

Figure 32: Comparison between state trajectories in the original set of OCPs (left) and with direct control from actor's predictions (right).

It can be noticed that state trajectories may be similar or completely different depending on the initial state. The policy suggested by the actor may drive the states towards different directions with respect to the original set of OCPs. As previously highlighted, velocity reaches high values without stabilizing or converging to a specific value.
It seems that the actor policy does not manage to stabilize the system and for some states it is not even able to minimize the cost.

9 Conclusions

In summary, this report presents the discussion of the learning process of a value function, the learning process of a policy π by minimizing the corresponding Q function, and using this policy to warm start an optimal control solver.
Using the single integrator as system model, results are sufficiently good, improving the system behaviour and making states and controls converge faster to the optimal values.
With the double integrator, instead, the learned policy is not satisfying, not being able to completely stabilize the system and minimize the cost.
In general, applying directly the policy suggested by the actor results, the system behaviour changes significantly and suggested controls show some sharp discontinuities, making it tricky to implement this method on a real system.
Suggesting actor policy to the OCP solver, instead, brings to smaller variations in system behaviour which are generally more feasible while pushing the solver in the right direction. This method helps the solver to move towards a more convenient minimum, adjusting its strategy (policy) to improve the overall performance, which is the objective of Reinforcement Learning.
Surely, there may be some potential improvements to this project to be implemented, especially for the double integrator case. For example, it is clear that there are some issues with the system stabilization. This may be due to issues in modeling or controlling the double integrator dynamics. Some possible solutions may be based on the following ideas:

- Tune hyperparameters: Experiment with different learning rates, discount factors, and neural network architectures to find a combination that stabilizes the learning process.
- Enhance exploration: Implement exploration strategies, such as epsilon-greedy policies, to properly implement exploration-exploitation trade-off.