Preprints of the 23rd IFAC World Congress
Busan, Republic of Korea, August 23-28, 2026

Citation: Javier Fañanas-Anaya, Gonzalo Lopez-Nicolas, and Carlos Sagues. Robot modeling with autoregressive physics-informed neural networks. 23rd IFAC World Congress Busan, Republic of Korea, August 23-28, pages 7768-7773, 2026

Robot Modeling with Autoregressive Physics-Informed Neural Networks

Javier Fañanás-Anaya * **Gonzalo López-Nicolás** *
Carlos Sagüés *

* *Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, Mariano Esquillor, 50018 Zaragoza, Spain (e-mail: javierfa@unizar.es, gonlopez@unizar.es, csagues@unizar.es)*

Abstract: Accurate and efficient system modeling is essential for control applications. Analytical models offer interpretability but often become unfeasible for complex systems, while data-driven neural networks require large datasets and may lack robustness. When system equations are known, Physics-Informed Neural Networks (PINNs) provide a powerful alternative by combining physics with data. In this work, we introduce AR-PINN, which addresses key limitations of PINN-based frameworks in control by handling time-varying control inputs and improving stability and accuracy over long prediction horizons. Simulations on a Two-Link Manipulator show that AR-PINN achieves high accuracy, robustness across different control scenarios, and a low computational cost.

Keywords: Grey box modeling, Physics-Informed Neural Networks, Data-driven modeling, Nonlinear system identification, Real-time simulation, Robot manipulator.

1. INTRODUCTION

Accurate system modeling is essential for simulation, state prediction, and control applications, see Schoukens and Noël (2017); Legaard et al. (2023). The challenge is particularly high in robotic systems due to the complex, nonlinear, and often unpredictable nature of their physics, see Fañanás-Anaya et al. (2024); Liu et al. (2024).

For control, it is essential to have a precise and computationally efficient model of the system see Forbes et al. (2015); Kordabad et al. (2023). If we know the equations of the system, it is natural to obtain an analytical or physics-based model. Analytical models are highly interpretable and robust, but if the equations are complex, solving them can be computationally costly or even unfeasible. Another drawback is that relying only on physical models can lead to inaccuracies in complex or unpredictable environments, where unmodeled dynamics or disturbances appear, see Mohajerin and Waslander (2019).

In the state-of-the-art, it is usual to find solutions based on Neural Networks (NN) for scenarios in which we need a computationally efficient solution, and flexibility in complex scenarios, see Gedon et al. (2021); Baby et al. (2022); Fañanás-Anaya et al. (2025). This solution requires a dataset, and the problems we may encounter are low interpretability and limited generalization capacity, as it will depend on the quality and variety of the dataset.

⋆ This work was supported via project REMAIN S1/1.1/E0111 (Interreg Sudoe Programme, ERDF) and projects CPP2024-011615, PID2021-124137OB-I00 and PID2024-159279OB-I00 funded by MICIU/AEI/10.13039/501100011033 and by ERDF/EU, and via Gobierno de Aragón project T45_23R.

When we know the constitutive equations of the system, but an analytical solution is not possible due to difficulty or efficiency, and we also have numerical data, an alternative state-of-the-art approach are Physics-Informed Neural Networks (PINNs), introduced by Raissi et al. (2019). PINNs require knowledge of the Ordinary Differential Equations (ODE) or Partial Differential Equations (PDE) of the system, as well as the initial and boundary conditions of the system. In addition, available real or simulated data from the system is included in the training. PINNs are more robust and have better generalization capabilities than a NN based only on data, in addition to obtaining lighter models and requiring smaller datasets, see Toscano et al. (2025). However, training a PINN is more complex than training a standard NN: combining physical information from ODEs or PDEs with data requires balancing multiple loss terms, and standard PINNs often struggle to converge compared to purely data-driven NNs.

1.1 Related Work

Standard PINNs allow us to approximate solutions to PDEs and ODEs. In the case of ODEs, we can define the PINN as a function f_p:

$$\hat{y}_t = f_p(t) \tag{1}$$

where $\hat{y}_t$ is the PINN estimation of the system state at time t. PINNs are trained and work correctly in a continuous time domain, where $t \in [0, T]$. Unlike numerical solvers, PINNs provide a direct mapping from time to state, meaning that the prediction at T can be obtained without explicitly computing intermediate states. This is a limitation in complex systems with a large time horizon T, as each state can be highly dependent on previous states. In such cases, the implicit temporal dependencies are not

©2026 the authors. Accepted by IFAC for publication under a Creative Commons Licence CC-BY-NC-ND

captured by the PINN, leading to a loss of accuracy when modeling long-term dynamics, see Wang et al. (2024). In addition, PINNs often struggle to generalize to time values larger than those seen during training, showing a rapid loss of accuracy for $t > T$. Another limitation of PINNs in the context of control, is that the standard PINN formulation (1) does not naturally incorporate control inputs.

To address these two main limitations of PINNs in the control context: lack of accuracy in long prediction horizons and support for control inputs, Antonelo et al. (2024) proposes PINC. The PINC framework is an extension of PINNs in which, we define function f_c:

$$\hat{y}_t = f_c(y_0, u_k, t) \tag{2}$$

where, in addition to the time t, the initial state y_0 and the control input u_k are also included. To address the lack of stability and accuracy over long prediction horizons, PINC introduces block prediction. Each block covers a time interval $t \in [0, T]$. If we consider a sequence of K blocks, we denote their indices by $k \in [1, 2, \ldots, K]$, and the total prediction horizon will be $K \cdot T$. In each block, y_0 corresponds to the final state $\hat{y}_T$ of the previous block and remains constant within the block, as does the control input u_k. The value of t changes within each block to obtain the state trajectory over the interval $[0, T]$. This results in an autoregressive prediction scheme, since the last predicted state $\hat{y}_T$ of one block becomes the initial state y_0 of the next.

The main limitations of PINC are that, although it is used autoregressively in the testing phase, this connection is not explicitly considered in the training phase. During training, each y_0 is randomly sampled rather than obtained from the previous block's prediction. This simplifies the training process and improves convergence, but may result in error accumulation over long prediction horizons, as PINC is not optimized to use its own predictions as input.

Another limitation is that PINC imposes the control inputs u_k to remain constant throughout each prediction block. This means that if the control signal changes frequently, the time interval duration T of each block must be small. Using smaller blocks requires more autoregressive connections, which in turn propagate and accumulate prediction errors. Furthermore, if the duration of each block is small, the learning capacity of PINNs is underutilized, limiting them to simple scenarios.

1.2 Contributions

We propose a new framework, the Autoregressive Physics-Informed Neural Network (AR-PINN). Our main contributions are: (i) AR-PINN maintains the block prediction of PINC, but introduces an autoregressive training scheme, improving accuracy over long prediction horizons; (ii) AR-PINN supports varying control inputs within each prediction block. This enables the use of larger time intervals, reducing the number of autoregressive steps and mitigating error accumulation across long prediction horizons. In addition, by being exposed to multiple control variations within a block, the network learns more complex dynamic behaviors of the system, leading to improved prediction accuracy and generalization; and (iii) we validate AR-PINN by modeling a Robotic Serial Manipulator System. The framework, the developed datasets, and the simulation results are publicly available [1].

2. PROBLEM STATEMENT

A wide range of dynamical systems can be described by ODEs or PDEs of the form:

$$\mathcal{F}(y, u, t, x, \frac{dy}{dt}, \frac{d^2y}{dt^2}, \ldots) = 0 \tag{3}$$

Where y denotes the system state, u the control input, and $\mathcal{F}$ is a differential operator that may involve derivatives of any order in time and space. ODEs are defined when the evolution depends only on time t, while PDEs describe systems whose dynamics depend on both time and spatial variables x. Given initial and, if required, boundary conditions, solving (3) yields the system trajectory.

For control applications, this continuous-time system is typically discretized. Let the state at discrete time t be $y_t \in \mathbb{R}^m$ and the control input be $u_t \in \mathbb{R}^n$, where the system dynamics are considered in discrete time over H prediction steps. The goal is to estimate the system states over H time steps $Y_{1,H} \in \mathbb{R}^{m \times H}$, given y_0 and the sequence of control inputs $U_{0,H-1} \in \mathbb{R}^{n \times H}$:

$$\begin{aligned} Y_{1,H} &= [y_1, y_2, \ldots, y_H] \\ U_{0,H-1} &= [u_0, u_1, \ldots, u_{H-1}] \end{aligned} \tag{4}$$

We define a function f that maps the initial state and the control sequence to the predicted states over the H-step horizon, resulting in $\hat{Y}_{1,H} \in \mathbb{R}^{m \times H}$:

$$\hat{Y}_{1,H} = f(y_0, U_{0,H-1}) \tag{5}$$

We propose our AR-PINN framework as f with the goal of minimizing the error between $\hat{Y}_{1,H}$ and $Y_{1,H}$.

3. PROPOSED FRAMEWORK: AR-PINN

We build from the block prediction strategy introduced in Antonelo et al. (2024). We denote by f_{ar} the function represented by our AR-PINN model. For an ODE system, we define the AR-PINN output for each block k as the estimation of the system states $\hat{Y}_k \in \mathbb{R}^{m \times n_t}$, given by:

$$\hat{Y}_k = f_{ar}(\hat{y}_{k-1}, U_{k-1}, t) \tag{6}$$

Where, $\hat{y}_{k-1} \in \mathbb{R}^m$ is the final state of the previous block (autoregressive input). $U_{k-1} \in \mathbb{R}^{n \times c}$ is the control sequence applied during block k, starting from the final state of the previous block $\hat{y}_{k-1}$, and consisting of c changes in the control signal:

$$U_{k-1} = [u^1_{k-1}, u^2_{k-1}, \ldots, u^c_{k-1}] \in \mathbb{R}^{n \times c} \tag{7}$$

Note that the notation for control inputs changes depending on the number of subscripts. When two subscripts are used, as in $U_{0,H-1} \in \mathbb{R}^{n \times H}$, the term represents the sequence of control inputs applied at each discrete time step of the prediction horizon. In contrast, with a single subscript, as in $U_{k-1} \in \mathbb{R}^{n \times c}$, it denotes the c control input changes applied within the block k.

The vector of the time instants within each block is defined as $t \in \mathbb{R}^{1 \times n_t}$, where n_t is the number of time steps per block, and its elements are arbitrarily sampled (not necessarily uniform) within the interval $[0, T]$. PINN-based

[1] https://github.com/javierfa98/AR-PINN

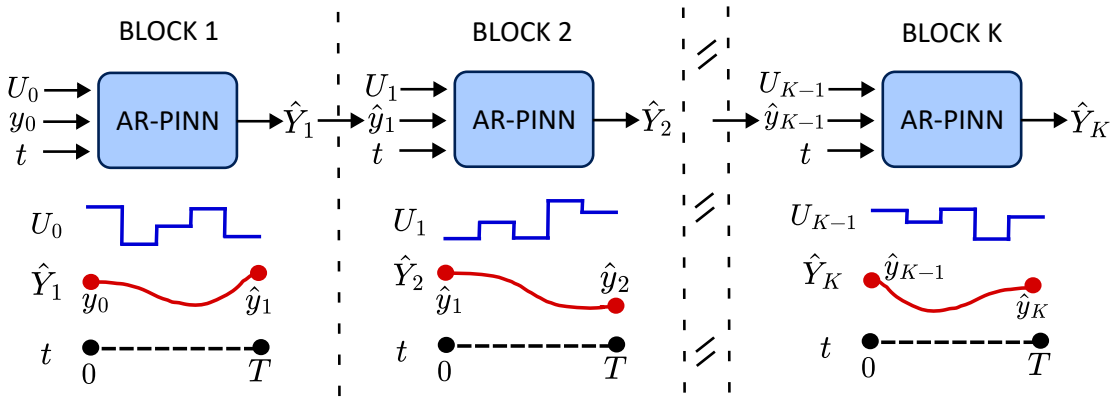


Fig. 1. Scheme of our framework with AR-PINN for K blocks. Note that to simplify the example, both the control signals U_k and the system states Y_k are represented with a single dimension ($n = 1, m = 1$). In the example, each block contains $c = 5$ control changes. In the first block, the AR-PINN inputs are the initial real state of the system y_0, the control inputs of the block from that initial state U_0, and n_t time instants from the interval $t \in [0, T]$. AR-PINN is evaluated n_t times to obtain all the solutions at the desired time instants, obtaining $\hat{Y}_1$ in block 1. The last prediction $\hat{y}_1$ of block 1, is the initial state used for the next block (autoregressive input). The remaining blocks follow the same procedure, with the initial state given by the last predicted state of the previous block.

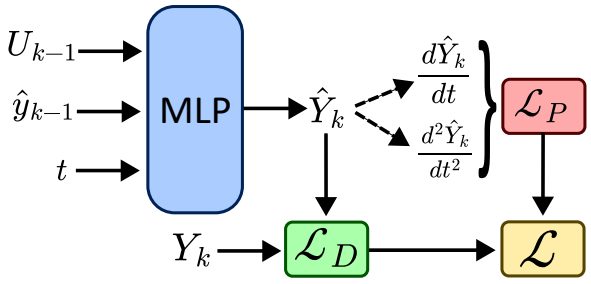


Fig. 2. Diagram of our AR-PINN architecture with the loss function for a block k. AR-PINN uses a Multi-Layer Perceptron to estimate the states $\hat{Y}_k$ of the system in block k, based on the final state $\hat{y}_{k-1}$ of the previous block $k-1$, the c changes in the control inputs U_{k-1} and the time interval t. From $\hat{Y}_k$ and its derivatives we compute the physical loss $\mathcal{L}_P$. $\mathcal{L}_D$ is computed from the initial and final states of the block and the reference values. Both losses leads to the total loss $\mathcal{L}$.

frameworks naturally operate in continuous time, but for control applications we discretize the time interval into n_t steps, enabling AR-PINN to predict the system state at each discrete instant.

Unlike PINC, which only supports constant control inputs per block, our AR-PINN allows variations of the control inputs within the same block. For this purpose, we split the time interval of the block into c subintervals of equal duration T/c (although intervals of different durations can also be considered), and assign a different control value to each subinterval. This capability is one of the key contributions of our framework: by allowing multiple control updates inside a single block, AR-PINN can operate with longer block durations without compromising stability. As a result, fewer autoregressive steps are required, which significantly reduces the accumulation of prediction errors over long horizons. Moreover, exposing the network to control changes within the same block enables it to capture richer and more complex system dynamics, improving model accuracy and generalization.

The output of the AR-PINN for block k is the estimated states over the block interval $\hat{Y}_k \in \mathbb{R}^{m \times n_t}$, defined as:

$$\begin{aligned} \hat{Y}_k &= [\hat{y}_k^1, \hat{y}_k^2, \ldots, \hat{y}_k^{n_t}] \in \mathbb{R}^{m \times n_t} \\ \hat{y}_{k-1} &= \hat{y}_k^1, \quad \hat{y}_k = \hat{y}_k^{n_t} \end{aligned} \tag{8}$$

The first element $\hat{y}_k^1$ of the block k, corresponds to the final state $\hat{y}_{k-1}$ of the previous block $k-1$, while the last element $\hat{y}_k^{n_t}$ represents the final state $\hat{y}_k$ of block k. The intermediate elements $\hat{y}_k^i$ capture the system evolution at each time step within the block, providing a complete representation of the state trajectory over the time vector.

Note that to obtain $\hat{Y}_{1,H}$ as in (5), we iterate K blocks of (6), since $H = K\, n_t$:

$$\hat{Y}_{1,H} = [\hat{Y}_1, \hat{Y}_2, \ldots, \hat{Y}_K] \tag{9}$$

Fig. 1 illustrates our proposed block prediction framework, showing the main parameters of each block, and the information transmission between blocks.

The architecture of the AR-PINN framework is a fundamental component of our approach, as it directly affects both model accuracy and computational efficiency. We propose a Multi-Layer Perceptron (MLP) architecture, based on PINN frameworks, which is significantly simpler than purely data-driven NN.

Following the training schemes of PINNs, the loss function used in the training algorithm includes a physical loss component $\mathcal{L}_P$ and a data-based loss component $\mathcal{L}_D$. A key contribution is that our proposed training algorithm updates the network weights with the chosen optimizer after each block prediction during training. AR-PINN, for block k, estimates the system states $\hat{Y}_k$. Using automatic differentiation, we compute the time derivatives required by the ODE or PDE governing the system. Note that these derivatives are computed from the network output rather than from measured data, making the approach robust to measurement noise. Based on these terms, the physical loss $\mathcal{L}_P$ is defined as:

$$\mathcal{L}_P = \text{MSE}(\hat{\mathcal{F}}, 0) \tag{10}$$

where MSE denotes the standard Mean Square Error, and $\hat{\mathcal{F}}$ is the physics residual estimated by AR-PINN.

The data-based loss component $\mathcal{L}_D$ is very flexible in PINN frameworks. Generally, initial states, boundary conditions, or trajectories of the real system that are known are included. However, since AR-PINN is a block prediction model, where the last state of a block is the initial state of the next block, we propose computing the data loss at both the initial and final states of each block. This requires obtaining the final state of each block prior to training, either through numerical integration or from real measurements. Thus, for a block k, we define $\mathcal{L}_D$ as:

$$\mathcal{L}_D = \text{MSE}(\hat{y}_{k-1}, y_{k-1}) + \text{MSE}(\hat{y}_k, y_k) \tag{11}$$

Note that both (10) and (11) illustrate the losses for a single block, but in practice these MSE terms are computed over multiple blocks in parallel, depending on the batch size. We define the total loss for each batch as:

$$\mathcal{L} = \lambda_P \mathcal{L}_P + \lambda_D \mathcal{L}_D \tag{12}$$

Where λ_P and λ_D are the values used to weigh $\mathcal{L}_P$ and $\mathcal{L}_D$. These weights balance the contribution so that neither dominates the optimization process and that both physical consistency and data fidelity are enforced during training. Fig. 2 shows a scheme of our AR-PINN architecture for a block k, and illustrates the computation of the losses.

Another key contribution is that we train AR-PINN using its autoregressive connection in the same way that it is later used in inference. In previous work on PINC Antonelo et al. (2024); Nicodemus et al. (2022), training does not explicitly incorporate this autoregressive dependency. The strategy used there consists of starting from a random initial state and minimizing the physical error in a single block, without using the autoregressive connection during training. This strategy simplifies the algorithm and facilitates convergence, obtaining models that predict with high accuracy within a block. However, when used in inference, each autoregressive connection introduces a small cumulative error, since the network has not been designed to use its own predictions as input. To mitigate this problem, we propose training directly with the autoregressive connection enabled, that is, training without teacher forcing, as in Fañanás-Anaya et al. (2025) and as illustrated in Fig. 1. This training strategy only affects $\mathcal{L}_D$, so it is essential to compute this metric at the most informative locations: in our case, at the beginning and end of each block.

4. MULTI-LINK MANIPULATOR

As an example of a real-world application, we consider a Multi-Link Manipulator, specifically the PowerCube Serial Robot, as described in Fehr et al. (2020), which provides equations and simulations to validate our model. In addition, this Two-Link Manipulator was also studied in Nicodemus et al. (2022) with the PINC framework, which enables a direct comparison between PINC and our proposed AR-PINN: qualitatively, by analyzing the modeling improvements introduced by AR-PINN, and quantitatively, by evaluating computational cost and long-term prediction accuracy.

The system is identified in the previously cited works with the following dynamics:

$$M(q)\ddot{q} + N(q, \dot{q}) = h(q, \dot{q}) + B\,u \tag{13}$$

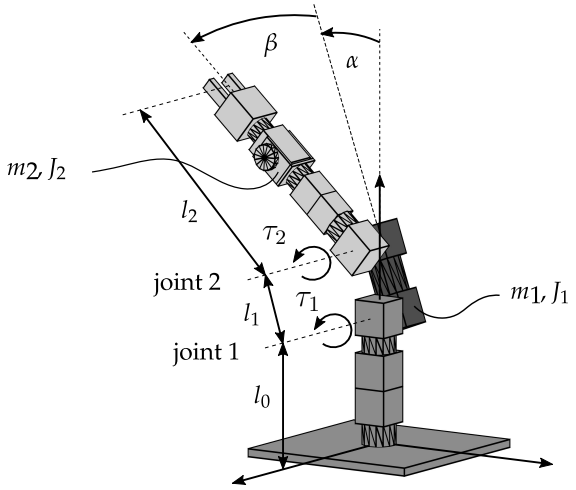


Fig. 3. Schematic of the Two-Link Manipulator. Adapted from Fehr et al. (2020).

where $q = [\alpha, \beta]^\top \in \mathbb{R}^2$ are the generalized coordinates, $M(q) \in \mathbb{R}^{2\times2}$ is the nonsingular mass matrix, $N(q, \dot{q}) \in \mathbb{R}^2$ is the vector of centrifugal, coriolis, and gyroscopic forces, $h(q, \dot{q}) \in \mathbb{R}^2$ is the vector of applied forces, $B \in \mathbb{R}^{2\times2}$ is the input matrix, and $u \in \mathbb{R}^2$ is the input vector, which are the motor currents of joint 1 and joint 2 (Fig. 3).

The system parameters include masses m_1 and m_2, inertias J_1 and J_2, link lengths l_0, l_1 and l_2. The motor torques τ_1 and τ_2 are linearly related to the currents I_1 and I_2, which are our control input: $u = [I_1, I_2]^\top$. The generalized coordinates are our system state: $y = q = [\alpha, \beta]^\top$.

Knowing the dynamics of this system, we can compute the function $\hat{\mathcal{F}}$ used in (10). For this second-order system, we calculate $\frac{d\hat{Y}_k}{dt}$ and $\frac{d^2\hat{Y}_k}{dt^2}$ from $\hat{Y}_k$ using automatic differentiation. Then we define $\hat{\mathcal{F}}$, based in (13) as:

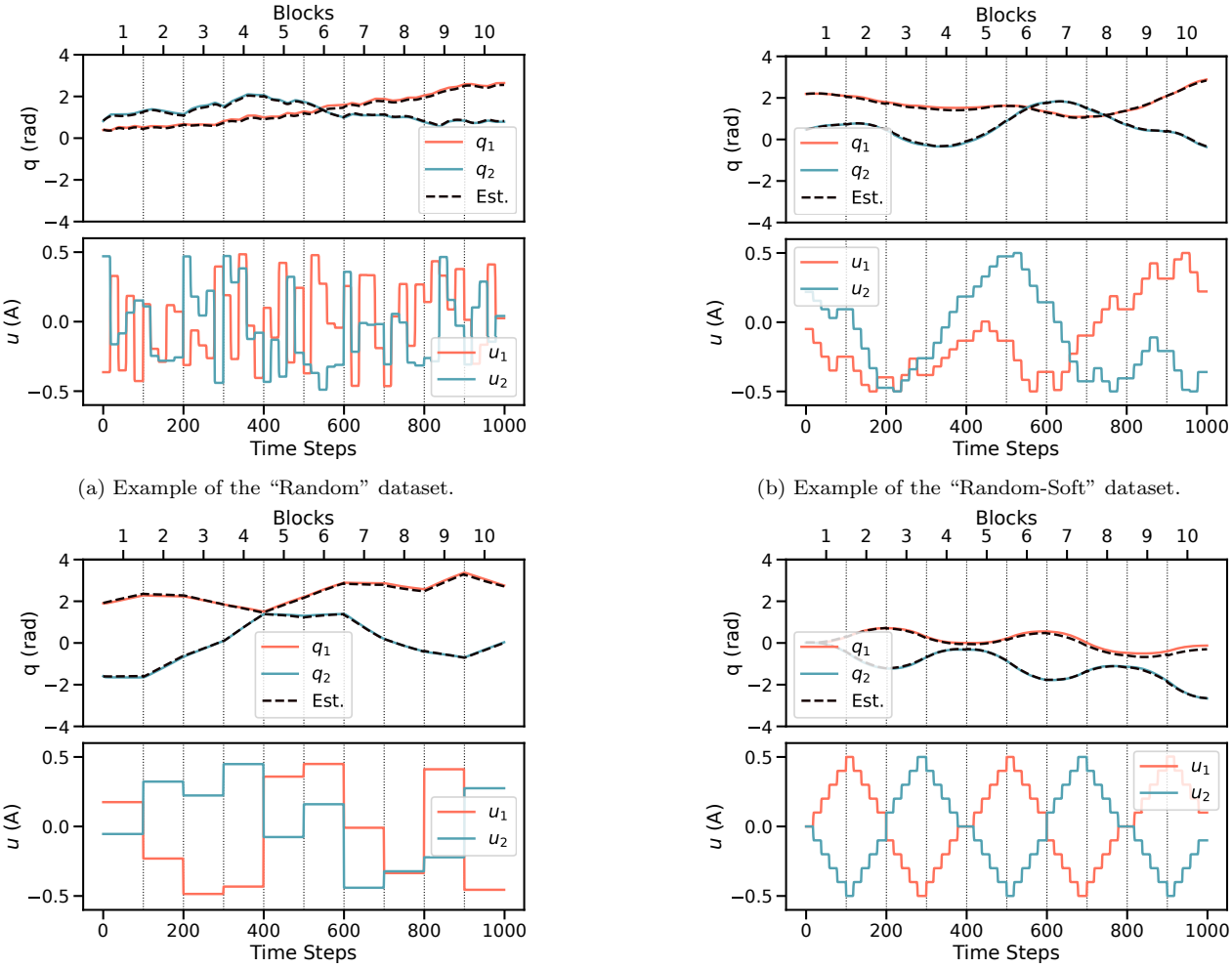
$$\hat{\mathcal{F}} = M(\hat{Y}_k)\frac{d^2\hat{Y}_k}{dt^2} + N(\hat{Y}_k, \frac{d\hat{Y}_k}{dt}) - h(\hat{Y}_k, \frac{d\hat{Y}_k}{dt}) - B\,U_{T(k-1),Tk} \tag{14}$$

5. SIMULATION RESULTS

For the simulations, we set the prediction horizon to $H = 10\,\text{s}$. Each prediction block covers $T = 1\,\text{s}$, resulting in $K = H/T = 10$ blocks. The control signal is updated every $0.2\,\text{s}$, yielding $c = 5$ control changes per block. We generate 3,000 simulations for training. Each simulation starts from a random initial state y_0, sampled component-wise from the interval $[-\pi, \pi]$. At every control update the input u is sampled component-wise from $[-0.5, 0.5]$.

To compute the data-based loss $\mathcal{L}_D$, we need the initial and final states of each prediction block. For this, we first generate the 3,000 simulations using an Euler integrator and store, for each block, its first and last state. This results in a total of 60,000 samples, a very small dataset compared to what would be required for a purely data-driven approach. During training, $\mathcal{L}_D$ uses only the initial and final states of each block, while $\mathcal{L}_P$ is evaluated at 100 collocation points randomly sampled within $t \in [0, 1]$.

We found the best results with an architecture consisting of 3 layers of 64 units, with hyperbolic tangent (tanh) as



(a) Example of the "Random" dataset.

(b) Example of the "Random-Soft" dataset.

(c) Example of the "Random-Constant" dataset.

(d) Experiment with a sinusoidal control input.

Fig. 4. AR-PINN estimation (Est), real state q and control inputs u in three examples from the test datasets, and for the experiment with a sinusoidal control input.

the activation function. We used the Adam optimizer for 2,000 epochs, where in the first 1,000 epochs we used a learning rate of 10^{-3}, and in the last 1,000 epochs we reduced the learning rate to 10^{-4}. Training ends with a second phase using the LBFGS optimizer, with the following hyperparameters: learning rate of 1.0, history size of 100, and 500 maximum iterations. The best results were obtained by normalizing inputs and states to $[-1, 1]$ and weighting the losses with $\lambda_P = 1$ and $\lambda_D = 2 \times 10^4$, ensuring both terms contribute on a comparable scale.

To test AR-PINN, we generated three test datasets, each containing 100 simulations, and an additional experiment with a sinusoidal control input. The "Random" dataset includes simulations where the control inputs are randomly generated at each change. It is important to note that these control signals do not correspond to realistic operating conditions; instead, they are intentionally designed as challenging scenarios to evaluate the robustness and generalization of the AR-PINN. The "Random-Soft" dataset generates smoother and more realistic control variations: the inputs are still random, but evolves gradually within each block, with bounded increments and occasional direction reversals, resulting in control trajectories that are smoother and closer to typical actuator behavior. Finally, the "Random-Constant" dataset consists of random control inputs that remain constant throughout each block.

We use the standard Mean Absolute Error (MAE) as the error metric during testing. To compute accuracy in the test scenarios, we evaluated AR-PINN in each simulation over the prediction horizon $H = 10$ s, with 10 blocks of $T = 1$ s. In each block, we divided the time uniformly with $n_t = 100$, so we can define t during test as:

$$t = [0, \Delta t, 2\Delta t, \ldots, T] \in \mathbb{R}^{1 \times n_t}, \quad \Delta t = \frac{T}{n_t - 1} \tag{15}$$

Table 1 shows the mean and standard deviation of the MAE for the 100 simulations of the test datasets. As can be seen, the results are very good, with very small MAE values, around 0.08 rads. The model shows consistent performance across all types of control inputs. The standard deviation is slightly smaller in "Random-Soft" and "Random-Constant", which is expected because "Random" is the most challenging scenario. However, even in that case, the model maintains good performance.

Fig. 4 shows the AR-PINN state estimation, the actual system state, and the control inputs in four examples,

Table 1. MAE and standard deviation of AR-PINN estimations.

	Random	Random-Soft	Random-Const
MAE (Rad)	0.0810	0.0790	0.0804
MAE Std (Rad)	± 0.0780	± 0.0437	± 0.0510

Table 2. Computational cost comparing PINC (Nicodemus et al. (2022)) and our AR-PINN.

	Parameters	Sim. Time (1s)	Euler/Sim. Time
PINC	12,994	$\approx 5.8 \times 10^{-3}$ s	55x
AR-PINN	9,346	$\approx 1 \times 10^{-3}$ s	320x

one for each test dataset and the experiment with a sinusoidal control input. The AR-PINN estimation is very accurate. By reducing the number of blocks, thanks to allowing several control inputs within each block, and by enabling the autoregressive connection during training, the estimates are consistent even over long prediction horizons such as these examples (1,000 time steps).

In Nicodemus et al. (2022), an experiment with a sinusoidal control input shows that PINC starts accumulating noticeable errors after second 4: at this time, their absolute error reaches ≈ 0.15 rad for q_1 and ≈ 0.3 rad for q_2. We replicated this experiment with AR-PINN and obtained, at second 4, absolute errors of 0.05 rad for q_1 and 0.004 rad for q_2. Furthermore, AR-PINN maintains consistency over the full 10 s horizon, with errors of 0.15 rad for q_1 and 0.007 rad for q_2 at the final time. Fig. 4(d) shows the AR-PINN estimation for 10 s under this experiment.

Table 2 shows a comparison of computational cost on an RTX 2060 GPU for the Two-Link Manipulator example between PINC (Nicodemus et al. (2022)) and our AR-PINN. The table reports the number of parameters, the execution time required to simulate 1 s of system evolution (with control input changes every 0.2 s), and the speedup relative to the Euler numerical integration baseline. AR-PINN achieves lower memory usage, faster execution time, and a significantly higher speedup. Note that the difference in execution time and speedup is mainly because AR-PINN, in addition to being a slightly smaller model in this example, with the configuration chosen in the simulation can calculate 1 s of simulation (5 control changes) with a single iteration of the network, parallelizing all time instants. PINC, on the other hand, must be executed once per control signal, resulting in 5 iterations of the network.

6. CONCLUSION

In this work, we introduced AR-PINN to address PINC limitations: AR-PINN enables multiple control updates within each prediction block, reducing the number of autoregressive steps and improving long-horizon stability. Furthermore, unlike PINC, AR-PINN is trained with its autoregressive connection, that is, without teacher forcing, which also improves long-horizon accuracy. In summary, AR-PINN enhances flexibility in handling control inputs and mitigates error accumulation, while being computationally efficient for real-time control applications.

REFERENCES

Antonelo, E.A., Camponogara, E., Seman, L.O., Jordanou, J.P., De Souza, E.R., and Hübner, J.F. (2024). Physics-informed neural nets for control of dynamical systems. *Neurocomputing*, 579, 127419.

Baby, T.V., Sotoudeh, S.M., and HomChaudhuri, B. (2022). Data-driven prediction and predictive control methods for eco-driving in production vehicles. *IFAC-PapersOnLine*, 55(37), 633–638.

Fañanás-Anaya, J., López-Nicolás, G., and Sagüés, C. (2024). Robot modeling with autoregressive transformers. In *7th Iberian Robotics Conference (ROBOT)*, 1–6.

Fañanás-Anaya, J., López-Nicolás, G., and Sagüés, C. (2025). Dynamical system simulation with attention and recurrent neural networks. *Neural Computing and Applications*, 37(4), 2711–2731.

Fehr, J., Schmid, P., Schneider, G., and Eberhard, P. (2020). Modeling, simulation, and vision-/MPC-based control of a PowerCube serial robot. *Applied Sciences*, 10(20).

Forbes, M.G., Patwardhan, R.S., Hamadah, H., and Gopaluni, R.B. (2015). Model predictive control in industry: Challenges and opportunities. *IFAC-PapersOnLine*, 48(8), 531–538.

Gedon, D., Wahlström, N., Schön, T.B., and Ljung, L. (2021). Deep state space models for nonlinear system identification. *IFAC-PapersOnLine*, 54(7), 481–486.

Kordabad, A.B., Reinhardt, D., Anand, A.S., and Gros, S. (2023). Reinforcement learning for MPC: Fundamentals and current challenges. *IFAC-PapersOnLine*, 56(2), 5773–5780.

Legaard, C., Schranz, T., Schweiger, G., Drgoňa, J., Falay, B., Gomes, C., Iosifidis, A., Abkar, M., and Larsen, P. (2023). Constructing neural network based models for simulating dynamical systems. *ACM Computing Surveys*, 55(11), 236:1–236:34.

Liu, J., Borja, P., and Santina, C.D. (2024). Physics-informed neural networks to model and control robots: A theoretical and experimental Investigation. *Advanced Intelligent Systems*, 6(5), 2300385.

Mohajerin, N. and Waslander, S.L. (2019). Multistep prediction of dynamic systems with recurrent neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 30(11), 3370–3383.

Nicodemus, J., Kneifl, J., Fehr, J., and Unger, B. (2022). Physics-informed neural networks-based model predictive control for multi-link manipulators. *IFAC-PapersOnLine*, 55(20), 331–336.

Raissi, M., Perdikaris, P., and Karniadakis, G.E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686–707.

Schoukens, M. and Noël, J.P. (2017). Three benchmarks addressing open challenges in nonlinear system identification. *IFAC-PapersOnLine*, 50(1), 446–451.

Toscano, J.D., Oommen, V., Varghese, A.J., Zou, Z., Ahmadi Daryakenari, N., Wu, C., and Karniadakis, G.E. (2025). From PINNs to PIKANs: recent advances in physics-informed machine learning. *Machine Learning for Computational Science and Engineering*, 1(1), 15.

Wang, S., Sankaran, S., and Perdikaris, P. (2024). Respecting causality for training physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 421, 116813.