# Fast initialization of control parameters using supervised learning on data from similar assets

Willems, Jeroen; Eryilmaz, Kerem; Steckelmacher, Denis; Depraetere, Bruno; Beck, Rian; Bey-Temsamani, Abdellatif; Helsen, Jan; Nowe, Ann

*Published in:*
2022 IEEE Conference on Control Technology and Applications, CCTA 2022

*DOI:*
https://doi.org/10.1109/CCTA49430.2022.9966037

*Publication date:*
2022

# Fast Initialization of Control Parameters using Supervised Learning on Data from Similar Assets

Jeroen Willems[1], Kerem Eryilmaz[1], Denis Steckelmacher[2], Bruno Depraetere[1], Rian Beck[1], Abdellatif Bey-Temsamani[1], Jan Helsen[3] and Ann Nowé[2]

***Abstract*— This paper proposes a method to provide a good initialization of control parameters to be found when performing manual or automated control tuning during development, commissioning or periodic retuning. The method is based on treating the initialization problem as a supervised learning one; taking examples from similar machines and similar tasks for which good control parameters have been found, and using those examples to build models that predict good control parameters for new machines and tasks yet to be initialized. Two of such models are proposed, one based on random forest regressors and a second based on neural networks. The random forest is highly data-efficient but generalizes only moderately. The neural network is able to leverage a high-dimensional *burner run* input to perform automatic system identification and generalization. While the proposed approach can be applied to a variety of applications for which example data from well functioning controllers can be used to hot-start new ones, we applied it in this paper to three slider-crank setups performing a variety of similar tasks. We found that both models outperform a benchmark of using a physics-inspired model for the initialization. Using 20% of the data for training, the required number of experiments was reduced up to 44%, and the performance of the initial experiments was improved by up to 68% compared to the benchmark.**

## I. INTRODUCTION

Modern trends such as mass-customization and flexible machine use, combined with an ever increasing demand for performance, result in a situation where it is no longer sufficient to tune a single controller once during development for a broad range of machines. Instead, it often becomes needed to tune or retune controllers for each specific unit built, or for each task, usage profile or set of conditions.

This tuning can be executed manually by operators, or by using automated methods such as auto-tuners [Leva et al., 2002], [Wang et al., 2008] and learning based approaches such as Reinforcement Learning [Li et al., 2019]. Both however typically require multiple experiments or an accurate simulator, and since tuning has to be done for several machines, tasks and / or conditions, the overall tuning time can quickly increase. This can cause long lead times and long commissioning processes, or prolonged periods of stalled production. Industry furthermore often avoids these issues by simply not tuning sufficiently or at all, as estimates for the US process industry for example are that only 1 in 3 controllers has an acceptable performance level, and that they could be tuned better but simply are not [Desborough and Miller, 2002].

To combat these issues, better tuning algorithms could be developed. Alternatively, like we propose in this paper, the tuning can be improved with a good initialization of the tuning process, as this will mean less tuning is needed afterwards. It will further also yield a better performance during the initial iterations, which can avoid machine damage or dangerous behavior during the tuning.

In literature, several methods are available to initialize on the basis of dynamic models describing the application's behavior. Such models are already often used for learning during the tuning process itself, but they can also be used for the initialization. However, while these methods are very robust in converging despite only having poor models [Bristow et al., 2006], they are not very good at the initialization itself. Recently some works have extended iterative learning control to multiple machines [Ronzani et al., 2020]. In this case, new machines and / or tasks can only be initialized well if (i) the model structure is really good (so most information learned is captured in the learned model parameters), or (ii) if the tasks are really similar. The first is typically not the case, as argued above, and the second limits the applicability of such methods (for initialization).

Other model-based learning methods such as [van de Wijdeven and Bosgra, 2010] employ basis functions, as they parameterize the control signal in terms of the task, such that the result can be extrapolated towards different tasks. These basis functions are derived based upon a model structure. In case these functions are selected correctly, the learned coefficients can be directly shared from one task to the next. However, as argued above, often only poor knowledge is available on the model structure, or no model at all, in which case these methods also yield a limited performance.

If dynamic models are not to be relied upon, data from similar tasks or conditions for which the control parameters have already been fine-tuned can be used for the initializing. In this context we denote tasks as similar if the needed control inputs for them have values that are close to each other, and systems as similar if the same control inputs lead to motions that resemble each other. For such cases, transformation-based approaches ([Janssens et al., 2012], [Willems et al., 2019]) exist, which aim to map the learned information from one task to the next. However, these methods do not generalize well towards different tasks lengths, and require sufficient task similarity.

[1]Flanders Make, Belgium
firstname.lastname@flandersmake.be
[2]Artificial Intelligence Lab, [3]Department of Mechanical Engineering, Vrije Universiteit Brussel, Brussels, Belgium
firstname.lastname@vub.be

In this paper, we will derive a framework for initializing, which we denote as hot-starting, employing machine learning-based methods, more specifically supervised learning. To do so, we re-use information of previous examples in the form of a labeled dataset containing a set of good control parameters and signals for a variety of different machines and / or tasks. On this dataset we train a model that is capable of hot-starting both parameters and signals for unseen tasks and / or systems. Since we consider similar systems and tasks, it is assumed that the statistical properties of the generative distribution of each system are the same. This approach is generically applicable and does not require a physical dynamic model. Concretely, we propose two algorithms for hot-starting, that differently balance the various requirements of industrial applications:

- **Direct approach**: using a data-efficient random forest. No dynamic model nor simulator is needed. The random forest directly predicts good control parameters for a new task and system, by generalizing from previously-seen tasks, systems and their good control parameters.
- **Burner-run approach**: the new system is first excited with *a burner run*, and its response, in addition to the task, is given to a neural network (compatible with such high-dimensional inputs) that predicts good control parameters. This approach does not need an identifier of the new system (the burner run performs implicit system identification), but is less data efficient than the direct approach. In this paper, we propose to use an approximate model (of which several parameters are varied) as a simulator, to generate enough (inexpensive) virtual data for the neural network to be trained on.

We experimentally demonstrate our two methods on a fleet of non-linear slider-crank setups. Both hot-starting methods aim to produce good initial control parameters, that are then fine-tuned for each task with an automated two-step Iterative Learning Control (ILC) algorithm [Volckaert et al., 2010]. We assess the quality of our algorithms by how much they reduce the required number of ILC iterations before convergence (hence saving time on the new system), as well as the performance increase of the first iteration. The proposed methods are compatible with any tuning algorithm (e.g., manual tuning), and make no assumption about the nature of the machines or the tasks being tuned. Therefore, the proposed approach can equally be applied to different applications for which example data from well functioning controllers can be used to hot-start new ones.

The remainder of this paper is organized as follows. Section II introduces the two-step ILC algorithm and the considered supervised learning algorithms. Section III introduces the fleet of slider-crank setups and their controllers. Section IV then outlines the considered hot-starting framework and the developed methods. Section V validates the proposed approach experimentally, and Section VI concludes the paper.

## II. PRELIMINARIES

### A. ILC algorithm

In this paper we make use of the generic two-step ILC approach presented in [Volckaert et al., 2010]. Its goal is to learn an optimal input signal $\mathbf{u} \in \mathbb{R}^{n_u \times N}$ (with $N$ time samples) to be applied to a generic nonlinear dynamical system $P$ (with $n_u$ control inputs and $n_y$ outputs), so that it behaves in the best sense possible. More specifically, it uses the applied inputs $\mathbf{u}_i \in \mathbb{R}^{n_u \times N}$ and observed outputs $\tilde{\mathbf{y}}_i \in \mathbb{R}^{n_y \times N}$ from iteration $i \in \mathbb{R}$, to decide on the control action $\mathbf{u}_{i+1}$ to apply during iteration $i+1$. To do so, it relies on an (approximate) model $\widehat{P}$ with the observed output $\tilde{\mathbf{y}} \in \mathbb{R}^{n_y \times N}$ and applied input $\mathbf{u}$, which is given by (using the lifted system representation):

$$\mathbf{y} = \widehat{P}\left(\mathbf{u}, \boldsymbol{\alpha}\right). \tag{1}$$

The correction terms $\boldsymbol{\alpha}$ are central in the ILC algorithm, and are explained further below.

Each ILC iteration $i$ consists of the following steps:

1) *Control step*: Given model correction terms $\boldsymbol{\alpha}_i$ (with $\boldsymbol{\alpha}_0 = 0$), calculate the input $\mathbf{u}_i$ that minimizes a given objective function $J$ satisfying constraints $g$:

$$\begin{aligned} \mathbf{u}_i = \arg\min_{\mathbf{u}} \;\; & J\left(\mathbf{u}, \mathbf{y}_i, \boldsymbol{\alpha}_i\right), \\ \text{s.t.} \quad & \mathbf{y}_i = \widehat{P}(\mathbf{u}, \boldsymbol{\alpha}_i), \\ & g\left(\mathbf{u}, \mathbf{y}_i, \boldsymbol{\alpha}_i\right) \leq 0. \end{aligned} \tag{2}$$

   If the objective is to follow a predefined reference $\mathbf{y}_r \in \mathbb{R}^{n_r \times N}$, the objective function can be chosen as $J(\mathbf{y}_i) = \|\mathbf{y}_r - \mathbf{y}_i\|_2^2$.

2) *Model correction step*: Calculate $\boldsymbol{\alpha}_{i+1}$ that minimizes the difference between the predicted output $\mathbf{y}_i$ and output $\tilde{\mathbf{y}}_i$ observed after applying input $\mathbf{u}_i$:

$$\boldsymbol{\alpha}_{i+1} = \arg\min_{\boldsymbol{\alpha}} \;\; \|\tilde{\mathbf{y}}_i - \widehat{P}(\mathbf{u}_i, \boldsymbol{\alpha})\|_2^2. \tag{3}$$

   Typically additional cost function terms are included to for example regularize $\boldsymbol{\alpha}$ or its changes w.r.t. $\boldsymbol{\alpha}_i$. Many different options can be used for the structure or parameterization of $\boldsymbol{\alpha}$, but in practice we usually use a combination of parametric corrections on the model parameters: $\boldsymbol{\alpha}_{\text{parametric}} \in \mathbb{R}^{n_p}$, as well as additive non-parametric correction signals: $\boldsymbol{\alpha}_{\text{non\_parametric}} \in \mathbb{R}^{n_a \times N}$ (an unparameterized vector, in this case added to the input), according to:

$$\mathbf{y} = \widehat{P}\left(\mathbf{u} + \boldsymbol{\alpha}_{\text{non\_parametric}}, \boldsymbol{\alpha}_{\text{parametric}}\right). \tag{4}$$

   This also explains what was written in the introduction; behavior that can be captured in the parametric model corrections can be mapped well to new cases, but since the model structure is never perfect, the learning will also rely on the additive non-parametric terms, and those can not efficiently be mapped to new cases.

### B. Supervised learning

Supervised learning is a research field that studies algorithms that learn from data mappings from inputs to outputs. Given a set of input-output pairs $\{\mathbf{x}, \mathbf{y}\}$ (with $\mathbf{x} \in \mathbb{R}^{n_x}$, $\mathbf{y} \in \mathbb{R}^{n_y}$), also called a *training set*, the objective of supervised learning algorithms is to produce a function $f \in \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$ such that $\hat{\mathbf{y}} = f(\mathbf{x})$ is as close as possible to $\mathbf{y}$, and still produce "good" $\hat{\mathbf{y}}$ outputs even for $\mathbf{x}$ inputs that have not been seen during training, but are drawn from the same distribution (called a *generalization*). Supervised learning can be seen as a form of function optimization, that finds the function $f$ that minimizes a *loss*, such as the prediction error. In that formalism, $f = \arg\min_f \mathcal{L}(f(\mathbf{x}), \mathbf{y})$.

In most applications of supervised learning, the inputs and outputs are vectors of real values (some applications, studied in the deep learning literature, consider more complicated inputs, such as images, sound waveforms, or text). With $\mathbf{y} \in \mathbb{R}^N$ and $\mathbf{x} \in \mathbb{R}^N$, the most commonly used loss is the Mean Squared Error (MSE) defined as $\mathcal{L}(f(\mathbf{x}), \mathbf{y}) \equiv \frac{1}{N}\sum_{k=1}^{N}(f(x(k)) - y(k))^2$.

While the inputs, outputs and loss are often straightforward to define in a generic way for supervised learning algorithms, what the function $f$ is, and how it is produced, depends on the particular algorithm being used. In this paper, we will consider two approaches, that we present later as we introduce our contributions. Random forests consider $f$ as a set of rules (a function with many if-then-else clauses in it), and compute the optimal set of conditions to check for based on the training set. Neural networks express $f$ as a sequence of matrix multiplications and non-linear activation functions (such as the hyperbolic tangent), in which learnable parameters appear, and iteratively compute the gradient of the loss $\mathcal{L}$ with regards to the parameters, and follow this gradient to tune the parameters so that $f$ leads to a progressively lower loss.

## III. USE CASE: FLEET OF SLIDER-CRANK MECHANISMS

### A. Slider-crank mechanism

In this paper, we consider the control of a fleet of slider-crank mechanisms, of which a single asset is shown schematically in Fig. 1.

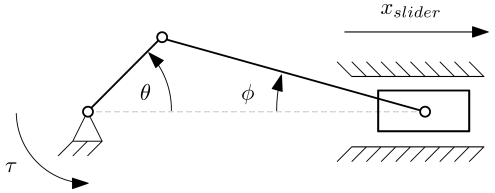



Fig. 1: Schematic overview of the slider-crank system.

The mechanism converts a rotary motion $y = \theta \in \mathbb{R}$ (with angular velocity $\dot{\theta} \in \mathbb{R}$) into a linear slider displacement $x_{\text{slider}} \in \mathbb{R}$ using input torque $u = \tau \in \mathbb{R}$ - a motion conversion that often emerges in industrial applications, such as weaving looms, compressors and piston engines. The system contains several challenging non-linearities, such as dead points and discontinuities (e.g., due to Coulomb friction and play). In the remainder of the paper, we use an approximate non-linear dynamical model $\widehat{P}$, which is described further in Appendix VI.

### B. Control objectives

The objective is to minimize the electrical losses $E \in \mathbb{R}$ over the control horizon with length $N$, denoted as:

$$\min_{\boldsymbol{\tau}, \boldsymbol{\theta}, \dot{\boldsymbol{\theta}}} \quad E = \boldsymbol{\tau}^T \boldsymbol{\tau}, \tag{5}$$

where $\boldsymbol{\tau} \in \mathbb{R}^N$ represents the torque signal. Additionally, the following motion constraints are taken into account:

$$\begin{cases} x_{\text{slider}}(k) \geq x_{\text{min}}, & k \in \{k_{\text{left}}, N - k_{\text{right}}\}, \\ \theta(1) = \pi, & \dot{\theta}(1) = 0, \\ \theta(N) = 3\pi, & \dot{\theta}(N) = 0. \end{cases} \tag{6}$$

The first constraint involves the height ($x_{\text{min}} \in \mathbb{R}$) and timing ($k_{\text{left}} \in \mathbb{R}$, $k_{\text{right}} \in \mathbb{R}$) of the displacement of the slider, as graphically represented in Fig. 2. The second and third constraint denote the initial and final angular conditions.

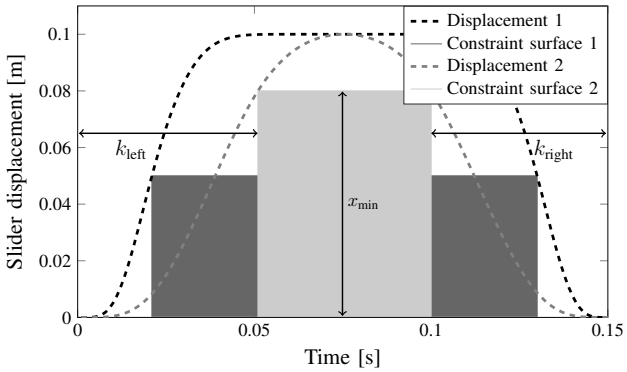



Fig. 2: Two different constraint surfaces with examples of corresponding feasible displacements.

The resulting cost (performance criterion) $\mathcal{J}_i \in \mathbb{R}$ for a given iteration $i$ is denoted as the weighted sum of the electrical losses $E_i$ and the (approximated and weighted) constraint violation for that iteration ($\boldsymbol{\nu}_i \in \mathbb{R}^4$):

$$\mathcal{J}_i = E_i + \gamma \sum \boldsymbol{\nu}_i. \tag{7}$$

In the above, $\gamma$ denotes a scalar constant and $\boldsymbol{\nu}_i$ is given by:

$$\boldsymbol{\nu}_i = \begin{bmatrix} |\theta_i(N) - 3\pi| \\ |\theta_i(N-1) - 3\pi| \\ 10\pi \max(0, x_{\text{min}} - x_{\text{slider},\,i}(k_{\text{left}})) \\ 10\pi \max(0, x_{\text{min}} - x_{\text{slider},\,i}(N - k_{\text{right}})) \end{bmatrix}. \tag{8}$$

Since the employed model $\widehat{P}$ contains model-plant mismatch w.r.t. the actual system $P$, we will use the ILC algorithm described in Section II-A to iteratively find the optimal solution to this problem. We will consider the learning as converged if the cost $\mathcal{J}_i$ differs less than 5% from its final value, and all entries of $\boldsymbol{\nu}_i$ (Eq. 8) are smaller than predefined tolerances ($2e^{-2}$).

The ILC looks for control solutions consisting of:

- The reference signal for the motor rotation $\boldsymbol{\theta}_{ref} \in \mathbb{R}^N$, which will be supplied to a feedback controller.
- The feedforward torque $\boldsymbol{\tau} \in \mathbb{R}^N$, which is parameterized using four parameters $\mathbf{p} \in \mathbb{R}^4$. It is found by applying linear interpolation over x-axis and y-axis breakpoints, defined as:

$$\mathbf{x} = \begin{bmatrix} 1 & k_{\text{left}} & k_{\text{right}} & N-1 & N \end{bmatrix}, \quad (9)$$

$$\mathbf{y} = \begin{bmatrix} p(1) & p(2) & p(3) & p(4) & 0 \end{bmatrix}. \quad (10)$$

An example is shown in Fig. 3.

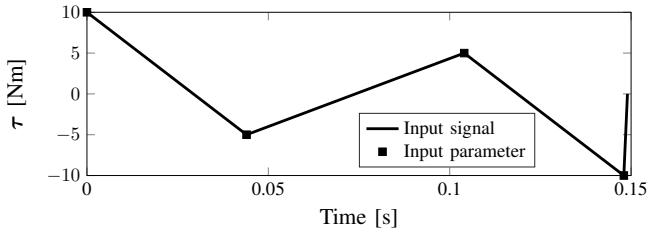



Fig. 3: Illustration of the considered input parametrization.

Hence, the proposed hot-starting framework will aim to hot-start reference signal $\boldsymbol{\theta}_{ref}$ and input parameters $\mathbf{p}$. The optimization problems are formulated in CasADi [Andersson et al., 2019] and are solved using IPOPT [Wächter and Biegler, 2006]. For the ILC correction terms $\boldsymbol{\alpha}$, we employ a non-parametric input correction.

### C. Experimental setup

We will experimentally apply the developed methods to a fleet of 3 slider-crank setups, of which one is shown in Fig. 4. These will be used for collecting data for training, as well as to validate the proposed methods afterwards. Each system is driven by a 3 $kW$ brushless servo motor with integrated drive. The lengths of the cranks and connecting rods are set to 0.05 $m$ and 0.3 $m$ respectively. The rotation of each motor is measured using a rotary incremental encoder (8192 CPR). The real-time control is executed by a Beckhoff real-time target running on Xenomai at a sampling frequency of 2000 $Hz$. Each system is controlled using a feedforward signal (motor torque $\boldsymbol{\tau} = \mathbf{u} \in \mathbb{R}^N$), as well as a feedback controller (PID) on the measured motor angle signal $\mathbf{y} = \boldsymbol{\theta} \in \mathbb{R}^N$, aiming to track reference signal $\boldsymbol{\theta}_{ref} \in \mathbb{R}^N$ in closed loop.

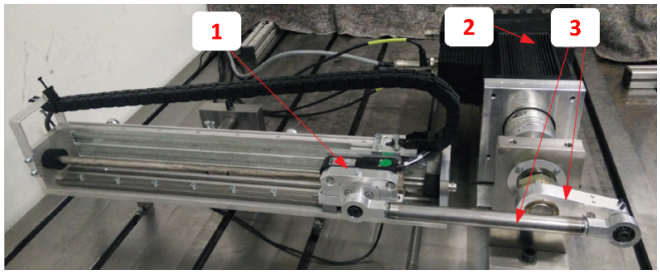



Fig. 4: A single slider-crank setup with its components: linear slider (1), rotary motor (2), the crank and rod (3).

### D. Considered tasks

We have three similar systems, and on each we will perform a range of similar tasks. All have a length of $N = 150$ samples (with $\Delta t = 0.001\ s$), but different $x_{\text{min}}$, $k_{\text{left}}$ and $k_{\text{right}}$ values, as shown in Fig. 5. In total, 24 tasks are considered for each of the 3 systems.

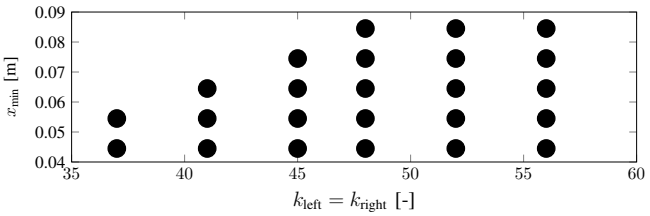



Fig. 5: The considered values for $x_{\text{min}}$, $k_{\text{left}}$ and $k_{\text{right}}$.

## IV. HOT-STARTING FRAMEWORK

The approach followed in this work is to treat the initialization as a generic supervised learning problem. In this approach examples are used, each example consisting of a system and a task to be executed as well as good control parameters for that combination. With those examples a machine learning model is trained that predicts the control parameters as a function of the system and the task. The task is expressed to the model as 3 values: $x_{\text{min}}$, $k_{\text{left}}$ and $k_{\text{right}}$. How the system is presented to the model is different in our direct and burner-run approaches, and is thus detailed below.

Once the models have been trained, they are then used when a new combination of system and task is needed, for which no set of optimal control parameters has yet been learned. They predict an initialization, after which the ILC algorithm described earlier is used to fine-tune the control parameters further until optimal values are found.

In the following sections, we will first detail the used data, and then the two types of models and how they are trained.

### A. Dataset generation and definition of benchmark

To generate training data, we let the ILC algorithm described above run, starting from a benchmark initialization. This was based on a simplified dynamic model $\widehat{P}$, with which the control step of the first ILC iteration is performed, but without any model corrections. After this initialization, we let the ILC gradually improve the control parameters until good performance is found. This procedure is performed for all combinations of tasks and systems, ending up with a dataset which contains for each task and system a good resulting set of control parameters, as well as the number of trials needed for the benchmark and the initial performance during the first trial, to compare the proposed methods to.

An example for system 1 is shown in Fig. 6. This figure shows the evolution of the inputs and outputs starting from the benchmark as well as the performance as it evolves over the iterations for a selected task ($x_{\text{min}} = 0.0745$, $k_{\text{left}} = k_{\text{right}} = 48$). During the first iteration there is a large constraint violation, but as the iterations progress, the constraint violation converges towards zero and the electrical

losses approach their steady state value, as visible in the combined cost function $\mathcal{J}$.

In this paper we used 20% of the data for training our initialization models, and the remaining for validation. We experimentally found that using less data for training resulted in poor model quality, whereas using more for training only yielded small improvements. To make things more realistic yet challenging, we further only use data from systems 2 and 3 for training, but none from system 1.

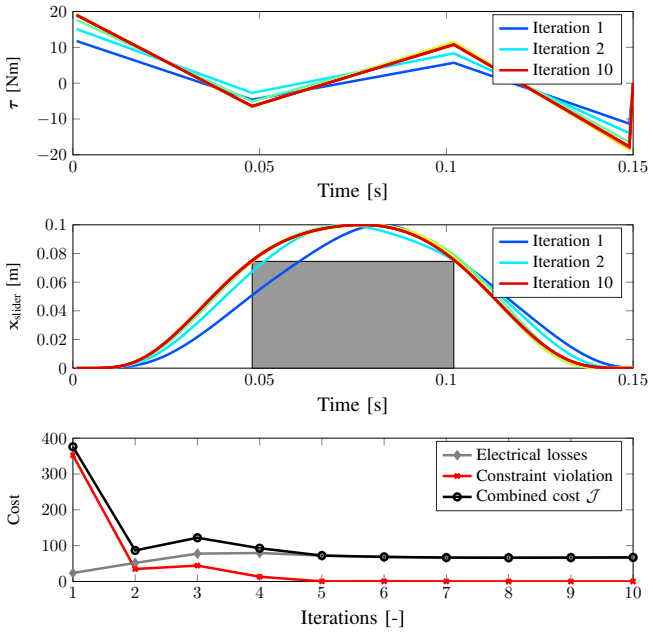



Fig. 6: Resulting motor torque $\boldsymbol{\tau}$, slider displacement $\mathbf{x}_{\text{slider}}$ and cost as a function of iterations.

### B. Direct approach (random forest)

The direct approach consists of building a random forest to solve the regression problem of mapping the selected task and the system identifier to the four control parameters. It tries to approximate the function $f$ such that $f(\mathbf{x}) = \mathbf{y}$ where $\mathbf{x} = \begin{bmatrix} x_{\text{min}} & k_{\text{left}} & k_{\text{right}} & s \end{bmatrix} \in \mathbb{R}^4$, and $\mathbf{y} = \hat{\boldsymbol{p}} \in \mathbb{R}^4$. $s$ can be a simple integer identifier, or a vector of one or more system parameters. The latter option provides better generalizability to unseen systems, while the former provides applicability to situations where system parameters are not known. For this study, we use simple integers as system identifiers to keep it maximally applicable.

A random regression forest is an ensemble model consisting of many (shallow) regression trees, whose outputs are averaged to get the final prediction. For a forest of size $J \in \mathbb{R}$, each tree $h_j$ (with $j \in [1, J]$) can be defined as $h(\mathbf{x}, \boldsymbol{\psi}_j)$ where $\mathbf{x}$ is the input vector, and $\boldsymbol{\psi}_j$ is a tree-specific configuration, consisting of a random subset of features $\mathbf{x}_j \subseteq \mathbf{x}$, as well as a tree structure consisting of split information. At each split in the tree, one of the features in $\mathbf{x}_j$ is compared to a constant threshold, determining the direction we traverse the next depth level in the tree. The leaves of the tree are constant values $\mathbf{y}_j$ which, in our case, correspond to estimates of the vector of parameters, i.e., $\hat{\boldsymbol{p}}$. The final output of the random forest is the average of all individual trees: $\frac{1}{J}\sum_{j=1}^{J} \mathbf{y}_j$. This is an example of combining weak learners to form strong learners: while the individual predictions of each tree are typically poor, the aggregate predictions can be quite accurate.

Building such trees is done using the CART algorithm [Breiman et al., 1984]. This algorithm recursively builds trees by choosing a feature and a threshold to split on at each level of the tree based on the reduction of variance such a split of the chosen training subsample produces, preferring the minimum variance in the two resulting subsets. The leaves are produced by averaging the parameter values in the resulting subsets from the split. The leaves are generated either when the maximum depth per tree is reached, or when the resulting subsets are below a certain size. The algorithm uses bootstrapping to randomly choose (with repetition) the subset of the training data to use for each tree being built.

In our case, the maximum depth was chosen as 5, the minimum number of samples to continue recursion was set as 2. In total, 50 trees were constructed to constitute the random forest. Due to the inherent variability of the training process, the training was repeated 10 times, and the model with the best out-of-bag (OOB) loss, or loss computed over the samples not used in the bootstrapped training process, was chosen as the final model. The loss chosen for this particular problem was the MSE described above.

For training, it is possible to use both real and simulated systems. In our case, we limited ourselves to using only the data from real systems, leaving simulated data to the burner run approach which can make better use of such data, as it has an explicit emphasis on system identification.

Once the forest is trained, using it is a trivial matter of providing the system identifier and task parameters $\begin{bmatrix} x_{\text{min}} & k_{\text{left}} & k_{\text{right}} & s \end{bmatrix}$ as input, and receiving the predicted $\hat{\boldsymbol{p}}$ directly as the average prediction from all 50 trees in the model. Conceptually, the model looks at various parts of the task description, as well as the system identifier, to produce progressively more specific estimates of the optimal parameters as we progress deeper into each tree. When we get to the leaf, we have our average prediction for the set of parameters and thresholds that the tree uses. Aggregating all these results by averaging, we get our final prediction. For previously seen systems, such an approach provides more system-specific estimates, while for unseen systems, it results in a prediction roughly averaged over all previously seen systems. Unless we use system parameters as system identifiers, this would give identical parameter estimates for all previously unseen systems. Since we are attempting to hot-start the ILC process, and not trying to obtain the final, optimal parameter values, such overgeneralizations do not necessarily constitute poor estimates.

The estimation of the reference signal works in a similar fashion. A random forest was built to map the resulting $\hat{\boldsymbol{p}}$ of the first random forest to the reference signal $\hat{\boldsymbol{\theta}}_{ref} \in \mathbb{R}^N$. During test time, the output of the first random forest is fed into the second one to get an estimate of the reference signal.

### C. Burner run approach (neural network)

The burner run approach performs hot-starting in two steps. First, the new system is excited with default control parameters which are chosen the same for every system. The response of the system $\mathbf{x}_{\text{slider}} \in \mathbb{R}^N$ is logged for $N = 150$ samples. Next, these readings are fed to a neural network, as are the 3 parameters to identify a task detailed earlier. Currently, the default control parameters correspond to the initial control parameters for a single task in the considered set of tasks. While this approach is capable of letting the neural network successfully perform the hot-starting, we note that more extensive design of experiment approaches can be used as well (to yield more exciting signals, potentially further improving performance), which we leave to future work. The neural network outputs control parameters $\hat{\mathbf{p}} \in \mathbb{R}^4$ and motor reference signal $\hat{\boldsymbol{\theta}}_{ref} \in \mathbb{R}^N$. We now briefly explain what neural networks are, what ours looks like, how it is trained, and how it is used for prediction.

Most current neural networks, and in particular the one we use in our burner run approach, are multi-layer perceptrons [Riedmiller, 1994]. A multi-layer perceptron considers an input in the form of a 1-dimensional vector of real values $\mathbf{x} \in \mathbb{R}^{n_x}$, and produces an output that is also a 1-dimensional vector of real values, $\hat{\mathbf{y}} \in \mathbb{R}^{n_y}$. The input and output do not need to have the same size. The neural network consists of a sequence of layers. Each layer $j \in \mathbb{R}$ takes as input the output of the previous layer $j-1$ (or $\mathbf{x}$ for the first layer), and produces an output $\mathbf{h}_j = \sigma(\mathbf{W}_j \mathbf{h}_{j-1} + \mathbf{b}_j) \in \mathbb{R}^{n_{h_j}}$, with $\mathbf{W}_j \in \mathbb{R}^{n_{h_j} \times n_{h_{j-1}}}$ a matrix of weights and $\mathbf{b}_j \in \mathbb{R}^{n_{h_j}}$ a vector of biases. Both these quantities are initialized to small random values when the network is created, and will be tuned as the network learns, so that its predicted output $\hat{\mathbf{y}}$ becomes as close as possible to the true outputs $\mathbf{y} \in \mathbb{R}^{n_y}$ in the training set. The activation function $\sigma$ can be any derivable non-linear function, such as the hyperbolic tangent, a sigmoid, or the rectified linear unit ($\sigma(\cdot) = \max(0, \cdot)$).

Defining how many layers a neural network should have, and what should be the size of each intermediate output $\mathbf{h}_j$, is task-specific and requires trial and error. In this article, we consider a neural network as shown in Fig. 7, with the sigmoid activation function. We observe that our neural network has two "legs", one that maps the burner run to a 16-dimensional intermediate value, and one that directly takes the 3 task parameters. This ensures that the burner run, that consists of many more values than the task parameters (150 vs. 3), does not take too much importance in the computation of the output of the network. Something similar happens on the output side: only 16 intermediate values are mapped to the 150-dimensional reference signal, to force the network to produce a control signal that is somewhat smooth. Extensive research exists which involves the optimization of the structure of neural networks and its hyper-parameters in an automated way, requiring thousands if not more of CPU or GPU-hours. Because we strive for a high level of applicability, we have not done so, but only tuned our neural network manually (without extensive hyper-parameter optimization). To obtain our neural network, we performed various manual experiments on a single computer with an AMD Ryzen 1700 processor (8 cores, 3.2 Ghz), 16 GB RAM. The manual experiments lasted for about two days in total. We therefore expect that any small or medium company is able to replicate our results and benefit from our insights, without requiring access to a supercomputer.

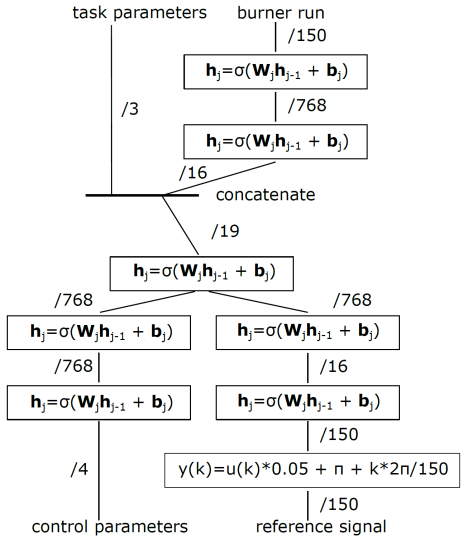



Fig. 7: The considered neural architecture. The "/N" notation indicates the number of values passing between components.

The neural network is trained with the Adam optimizer [Kingma and Ba, 2014] to minimize the MSE between its outputs (the control parameters and reference signal) and the ground-truth outputs, in the training set. The training set is obtained by combining two sources of training data:

- A *real* dataset obtained on the physical setup, for a specific (low) number of tasks. This allows, for the real machines, to have a mapping between task descriptions and optimal control parameters and reference signals. We also produced 3 burner run responses, one per physical machine. This allows to fully train our neural network, that takes as input the task parameters and a burner run, and outputs the predicted control parameters and reference signal.
- A *simulated* dataset found by optimizing the control parameters and reference signal on a simulator of our setup. Because no simulator is fully accurate, we embrace this inaccuracy and further push it by randomizing the simulator: every time it is used, some model parameters (arm length, gravity, friction, torque, ...) are perturbed with normal noise. Because each use of the simulator is *de facto* a different machine, we produce one distinct burner run per simulated training point.

The use of a randomized simulator, with burner runs, forces the neural network to learn to look at the burner run, and to discover how the burner run allows to know what control parameters and reference signal are good for a machine. We thus force it to learn to implicitly perform system identification.

Once trained, the hot-starting is performed as follows:

1) If the machine is new, perform a burner run on it, to obtain $\mathbf{B}_m \in \mathbb{R}^N$ (the burner run of machine $m$). If the machine is known, but only the task is new, then retrieve $\mathbf{B}_m$ from a list of known burner runs.
2) Feed $\mathbf{B}_m$ and the task parameters to the neural network. It produces the control parameters $\hat{\mathbf{p}}$ and the reference signal $\hat{\boldsymbol{\theta}}_{ref}$ described in Section III.
3) Use these predictions as hot-start for the ILC, which fine-tunes the control parameters and reference signal.

## V. EXPERIMENTAL RESULTS

### A. Experimental validation

As stated earlier, we have trained both hot-starting methods using 20% of the data, using a random selection of 7 tasks on systems 2 and 3 each (totaling $14/72$). Afterwards, we have hot-started all tasks of system 1 and the missing tasks for systems 2 and 3.

An example for a single task, equal to the one shown in Fig. 6, is shown in Fig. 8. Both the hot-starting approaches outperform the benchmark method. The first iteration of the slider displacement for the benchmark case violates the constraints significantly, whereas this is reduced for both the hot-starting approaches. Furthermore, the initial cost is significantly closer to its converged value when hot-starts are applied, indicating less further learning is required (reduced transients). For this task, the benchmark method requires 7 iterations to converge, the direct approach requires 4 iterations, and the burner-run approach requires 3 iterations.

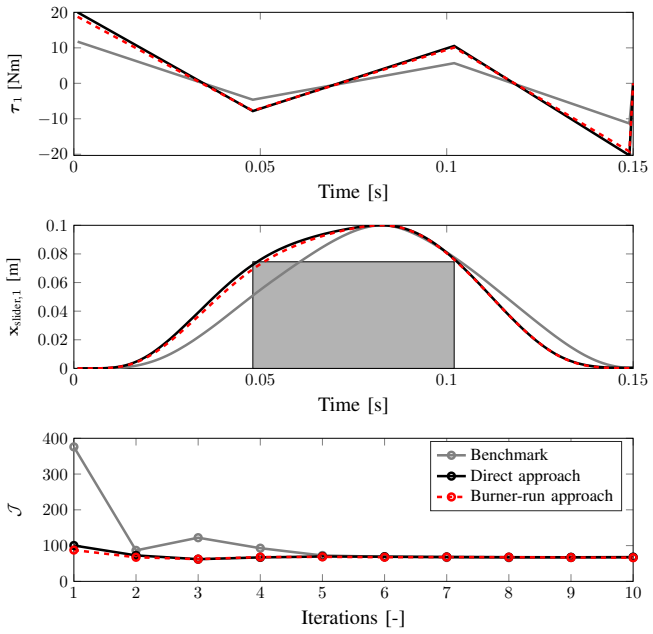



Fig. 8: Resulting $\boldsymbol{\tau}_1$, $\mathbf{x}_{\text{slider},1}$ ($1^{st}$ iteration) and $\mathcal{J}$ (all iterations) for the benchmark and two hot-starting approaches.

The performance statistics for all the tasks in the validation set for both of the hot-starting approaches are shown in Table I.

The following observations are made:

- **Iterations saved**: the direct approach saves approximately 21% iterations w.r.t. the benchmark, and the burner-run approach saves 44% of iterations, with up to 64% for system 3.
- **Initial motion improvement**: we calculate the 2-normed difference between the first and last slider displacement, i.e., $||\mathbf{x}_{\text{slider, 1}} - \mathbf{x}_{\text{slider, 10}}||_2$ and compare it to the benchmark, where average improvements of 52 - 68% are achieved. This indicates that the initial slider displacement is closer to its optimum. The initial constraint violation is further reduced, and the initial cost is also a lot lower (and closer to its value after convergence) with the hot-starting methods.

TABLE I: Performance of both hot-starting approaches.

| | **Sys. 1** | **Sys. 2** | **Sys. 3** | **Avg.** |
|---|---|---|---|---|
| **Direct: iterations saved** | 25% | 2% | 37% | 21% |
| **Direct: initial motion** | 53% | 43% | 61% | 52% |
| **Burner-run: iterations saved** | 42% | 26% | 64% | 44% |
| **Burner-run: initial motion** | 62% | 68% | 75% | 68% |

### B. Discussion

Both approaches outperform the benchmark, yielding a significant improvement in learning time and initial performance. The direct approach however performs slightly worse than the burner-run one, and also more variable, with in some cases yielding hardly any improvement over the benchmark.

As a result, based on the current findings the burner-run approach seems preferable, if a large dataset is available or simulation models that allow to extend the dataset. If neither of those are present, the direct approach is favorable. Furthermore, the training process of the direct approach is relatively simpler compared to the burner-run approach.

We also compared both approaches with the input transformation-based initialization from [Willems et al., 2019]. In terms of hot-starting the feedforward signal $\boldsymbol{\tau}$, similar gains were achieved as for the burner-run approach.

## VI. CONCLUSIONS

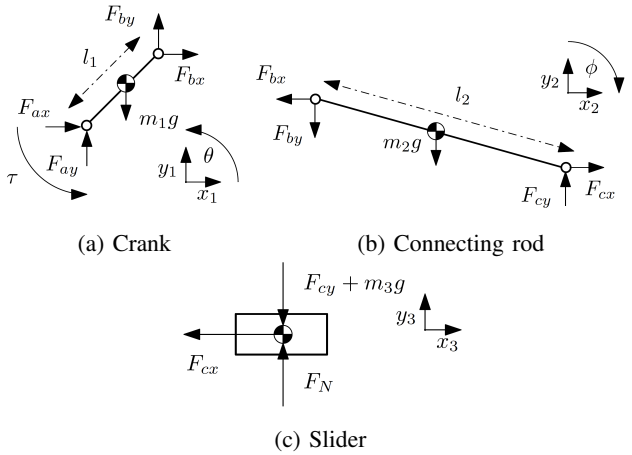
This paper presented methods for fast initialization of control parameters. For 3 slider-crank systems, we experimentally found a 21 - 44% reduction in tuning time, and an initial performance 52 - 68% better than with a model-based benchmark, using 20% of the data for training.

The developed methods can be used to speed up development, commissioning, or regular retuning. Especially for systems performing a multitude of similar tasks this improvement can be significant. While this paper considered 3 similar systems performing a small set of similar tasks, the authors expect the methods to be relevant for many possible scenarios: other types of systems, a larger variety of task variations, different operating conditions, different machine releases or variants, different software versions, etc. Future work can tackle those cases, and especially cases with bigger

differences. Expected is that then at some point multiple hot-starting models will become needed, each covering a part of the possible combination space. Then the burner run approach with its implicit identification will be critical to decide when which model should be used, or even when the differences are so large that the proposed initialization methods won't be beneficial at all.

## APPENDIX

The considered slider-crank system is modeled using a multi-body diagram [De Groote et al., 2021], see Fig. 9.




(a) Crank (b) Connecting rod

(c) Slider

Fig. 9: Multi-body diagram of the slider-crank system.

The corresponding equations of motion are then given by:

$$\begin{aligned}
m_1\ddot{x}_1 &= F_{ax} + F_{bx} \\
m_1\ddot{y}_1 &= F_{ay} + F_{by} - m_1 g \\
(J_1 + J_m)\ddot{\theta} &= \tau - b_m\dot{\theta} - 2r_{1_x}F_{bx} + 2r_{1_y}F_{by} - r_{1_y}m_1 g \\
m_2\ddot{x}_2 &= -F_{bx} + F_{cx} \\
m_2\ddot{y}_2 &= -F_{by} + F_{cy} - m_2 g \\
J_2\ddot{\phi} &= -r_{2_x}F_{bx} - r_{2_x}F_{cx} - r_{2_y}F_{by} - r_{2_y}F_{cy} \\
m_3\ddot{x}_3 &= -F_{cx} - b_s\dot{x}_3 \\
0 &= -F_{cy} + F_N - m_3 g
\end{aligned} \tag{11}$$

In the above, $m_1$, $m_2$ and $m_3$ denote the masses of the crank, connecting rod and slider respectively, and $J_1$, $J_2$ and $J_m$ denote the inertia of the crank, rod and motor. Furthermore, $\tau$ denotes the motor input torque, and $b_m$ and $b_s$ denote the damping constants of the motor and slider. $l_1$ and $l_2$ are the lengths of the crank and rod, and we define $r_{1_x} = \frac{1}{2}l_1 sin(\theta)$, $r_{1_y} = \frac{1}{2}l_1 cos(\theta)$, $r_{2_x} = \frac{1}{2}l_2 sin(\phi)$, $r_{2_y} = \frac{1}{2}l_2 cos(\phi)$. The slider position $x_{\text{slider}}$ (equal to $x_3$) is defined as $x_{\text{slider}} = l_1 cos(\theta) + l_2 cos(\phi) + l_1 - l_2$, employing geometric constraint $\phi = sin^{-1}(\frac{l_1}{l_2}sin(\theta))$.

Next, we define the state vector $\mathbf{x} = \begin{bmatrix}\theta & \dot{\theta}\end{bmatrix}^T \in \mathbb{R}^2$, input $u = \tau \in \mathbb{R}$ and correction terms $\alpha$ (parametric and / or non-parametric). Then, we can write the system in state-space format:

$$\begin{aligned}
\dot{\mathbf{x}} &= f(\mathbf{x}, u, \alpha), \\
y &= h(\mathbf{x}, u, \alpha) = \theta,
\end{aligned} \tag{12}$$

where $y \in \mathbb{R}$ denotes the output. The non-linear function $f$ is obtained by solving Eq. 11 symbolically using CasADi [Andersson et al., 2019]. The resulting model can then be denoted using the lifted representation as: $\mathbf{y} = \widehat{P}(\mathbf{u}, \boldsymbol{\alpha})$, with $\mathbf{y} \in \mathbb{R}^N$, $\mathbf{u} \in \mathbb{R}^N$ and $\boldsymbol{\alpha}$ the correction terms (parametric and / or non-parametric), given signal length $N$.

## ACKNOWLEDGMENT

This research received funding from the Flemish Government (AI Research Program). It was also supported by Flanders Make's SBO project 'MultiSysLeCo' (Multi-System Learning Control), funded by the agency Flanders Innovation & Entrepreneurship (VLAIO) and Flanders Make. Flanders Make is the Flemish strategic research centre for the manufacturing industry.

## REFERENCES

[Andersson et al., 2019] Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M. (2019). CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36.

[Breiman et al., 1984] Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). Cart. *Classification and Regression Trees; Wadsworth and Brooks/Cole: Monterey, CA, USA*.

[Bristow et al., 2006] Bristow, D. A., Tharayil, M., and Alleyne, A. G. (2006). A survey of iterative learning control. *IEEE Control Systems*, 26(3):96–114.

[De Groote et al., 2021] De Groote, W., Kikken, E., Hostens, E., Van Hoecke, S., and Crevecoeur, G. (2021). Neural network augmented physics models for systems with partially unknown dynamics: Application to slider-crank mechanism. *IEEE/ASME Transactions on Mechatronics*.

[Desborough and Miller, 2002] Desborough, L. and Miller, R. (2002). Increasing customer value of industrial control performance monitoring-honeywell's experience. In *AIChE symposium series*, number 326, pages 169–189. New York; American Institute of Chemical Engineers; 1998.

[Janssens et al., 2012] Janssens, P., Pipeleers, G., and Swevers, J. (2012). Initialization of ilc based on a previously learned trajectory. In *American Control Conference (ACC)*, pages 610–614. IEEE.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Leva et al., 2002] Leva, A., Cox, C. J., and Ruano, A. E. (2002). Hands-on pid autotuning: a guide to better utilisation.

[Li et al., 2019] Li, Y., Wen, Y., Tao, D., and Guan, K. (2019). Transforming cooling optimization for green data center via deep reinforcement learning. *IEEE transactions on cybernetics*, 50(5):2002–2013.

[Riedmiller, 1994] Riedmiller, M. (1994). Advanced supervised learning in multi-layer perceptrons—from backpropagation to adaptive learning algorithms. *Computer Standards & Interfaces*, 16(3):265–278.

[Ronzani et al., 2020] Ronzani, D., Steinhauser, A., and Swevers, J. (2020). Multi-system iterative learning control: an extension of ilc for interconnected systems. In *2020 IEEE 16th International Workshop on Advanced Motion Control (AMC)*, pages 79–84. IEEE.

[van de Wijdeven and Bosgra, 2010] van de Wijdeven, J. and Bosgra, O. H. (2010). Using basis functions in iterative learning control: analysis and design theory. *International Journal of Control*, 83(4):661–675.

[Volckaert et al., 2010] Volckaert, M., Swevers, J., and Diehl, M. (2010). A two step optimization based iterative learning control algorithm. In *ASME 2010 Dynamic Systems and Control Conference*, pages 579–581. American Society of Mechanical Engineers.

[Wächter and Biegler, 2006] Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25:57.

[Wang et al., 2008] Wang, Y.-B., Peng, X., and Wei, B.-Z. (2008). A new particle swarm optimization based auto-tuning of pid controller. In *2008 International Conference on Machine Learning and Cybernetics*, volume 4, pages 1818–1823. IEEE.

[Willems et al., 2019] Willems, J., Hostens, E., Depraetere, B., Ronzani, D., Steinhauser, A., and Swevers, J. (2019). An input-mapping initialization approach to accelerate iterative learning of similar tasks. In *Proceedings of IEEE 2019 International Conference on Mechatronics*.