Online Learning of the Inverse Dynamics with Parallel Drifting Gaussian Processes: Implementation of an Approach for Feedforward Control of a Parallel Kinematic Industrial Robot

Tim-Lukas Habich, Daniel Kaczor, Svenja Tappe and Tobias Ortmaier
Institute of Mechatronic Systems, Gottfried Wilhelm Leibniz University, 30167 Hanover, Germany
tim-lukas.habich@stud.uni-hannover.de, daniel.kaczor@imes.uni-hannover.de

***Abstract*—The present paper deals with an online approach to learn the inverse dynamics of any robot. This is realized by the use of Gaussian Processes drifting parallel along the system data. An extension by a database enables the efficient use of data points from the past. The central component of this work is the implementation of such a method in a controller in order to achieve the actual goal: the feedforward control of an industrial robot by means of machine learning. This is done by splitting the procedure into two threads running parallel so that the prediction is decoupled from the computing-intensive training of the models. Experiments show that the method reduces the tracking errors more clearly than an elaborately identified rigid body model including friction. For a defined trajectory, the squared areas of the tracking errors of all axes are reduced by more than 54% compared to motion without pre-control. In addition, a highly dynamic pick-and-place experiment is used to investigate the possible changes in system dynamics. Compared to an offline trained model, the approximation error of the proposed online approach is smaller for the remaining time of the experiment after an initial phase. Furthermore, this error is smaller throughout the experiment for online learning with parallel drifting Gaussian Processes than when using a single one.**

***Index Terms*—online learning, inverse dynamics, feedforward control, implementation, Gaussian Process**

I. Introduction

A torque pre-control is often used in the drive control of robots to improve the command input response. From the desired values in the joint space, i.e. joint angles $\boldsymbol{q}_\mathrm{d}$, joint velocities $\dot{\boldsymbol{q}}_\mathrm{d}$ and joint accelerations $\ddot{\boldsymbol{q}}_\mathrm{d}$, the required feedforward torques $\boldsymbol{\tau}_\mathrm{FF}$ are determined using the inverse dynamics model. Since this model is not exact, the actual values $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$ deviate from the desired values. The difference between these, the tracking error, has to be compensated by the controller afterwards. Hence, the quality of the inverse dynamics model is crucial, since a more accurate model reduces the tracking error and thus relieves the controller.

Conventional methods for modeling dynamics have the disadvantage that all system properties, such as friction in the joints, have to be modeled. In practice, this turns out to be a complex task and can still be imprecise despite the high modeling effort. In addition, all model parameters are determined offline and are therefore assumed to be constant. Thus, changes in the robot structure that occur during operation cannot be taken into account. For these reasons, a new approach has been established: *the online learning of the inverse dynamics*. Methods in this category model robot dynamics with the aid of measurement data. Therefore, physical system properties are automatically taken into account in the model. Online learning methods also have the property that the model adapts permanently to new data. When the system dynamics change, e.g. due to an additional mass at the end effector in a simple pick-and-place process, the model is gradually updated. Thus, the changed dynamics are *learned* after a few control cycles. In addition, the time-consuming recording and training on a sufficiently large training data set required for offline learning is no longer necessary, since the model is determined during robot operation and continuously adapted to the data.

Due to these positive aspects, a large number of authors have researched the field of online model learning in recent years in order to develop algorithms that are real-time capable. Real-time capability means in this connection that the algorithm has to reliably provide the feedforward moments $\boldsymbol{\tau}_\mathrm{FF}$ for each new control cycle based on the current model and the desired values in the joint space. Since typical cycle times of controllers are in the range of 1 ms - 5 ms, this requirement for real-time capability proves to be a considerable boundary condition.

A comparison of popular approaches to learning inverse dynamics has already been made [1]. Gaussian Process Regression (GPR) [2] and ν-Support Vector Regression (ν-SVR) [3] have a higher accuracy than Locally Weighted Projection Regression (LWPR) [4]. However, it was found that LWPR is significantly more computationally efficient than the other two methods. Hence, this algorithm is suitable for online learning of inverse dynamics in real-time. The combination of the computing efficiency of LWPR and the accuracy of GPR or ν-SVR in one approach turned out to be a central goal. In the past, both local and sparse methods were used to accelerate the process. Local online SVR was introduced for real-time learning for low-level robot controls, whereby only local data points near the operation point are used for training [5]. The performance was only slightly worse compared to the usual SVR approach, as the local data points used are much more important for the learning performance than data points that are far away from the operation point [6]. Local Gaussian Processes have been proposed as another local approach to real-time online model learning [7]. The training data is divided into local regions so that an individual Gaussian Process (GP) model can be trained for each region. Thus, the prediction for a query point is made by the weighted estimation with nearby local GP models.

In contrast to local containment, the spectrum of the kernel matrix can be sparsified with sparse methods which was used in the development of the real-time capable Incremental Sparse Spectrum GPR algorithm [8]. Besides, it is common with Sparse Gaussian Process Regression (SGPR) to determine a small number of data points which are used to approximate the kernel matrix. SGPR has already been used for online learning of inverse dynamics in which several Gaussian Processes drift parallel along the data [9]. Each GP model has a defined size. Thus, after the startup, the number of data points from the *past* in each GP model remains constant. By startup we mean the filling of the GP models with data points up to the maximum size. Due to the various sizes, the predictions of the models are also different. Purely intuitively one would say that a model with more data points also provides a higher accuracy. This is not always the case. Depending on the curve section, either a smaller or a larger model can approximate better [9]. By appropriate weighting the individual predictions, this variable accuracy can be taken into account in the final prediction.

In addition to the sparseness property due to the use of SGPR, this approach is also local in a certain sense because the GP models only contain a defined number of data points from the past. The oldest data point is replaced in each step by the newest point and is therefore no longer used. However, if you come to an area in which you were in the past, it would make sense to use this data again. Such a storage of old points in an online learning procedure has already been presented and it has been shown that the performance is significantly improved by a database [10]. A certain number of nearest neighbors from the newest data point were added to the training set in order to make use of the existing knowledge in the current region.

The present work deals with the fusion of the latter two approaches: the extension of the parallel drifting Gaussian Processes by a database. In section II, the SGPR is presented and the entire concept of the used procedure for pre-control is introduced. In addition, it is examined whether the method is real-time capable. It will become apparent that this is not the case due to the computing time for one run which takes too long. How such a procedure can nevertheless be implemented in robot control is a central component of this paper and is described in detail in section III. Based on this, in section IV the developed pre-control is experimentally validated with a parallel kinematic industrial robot. In section V conclusions are drawn and a final outlook for future tasks is given.

II. Parallel Gaussian Processes for Online Learning

In this section, the online learning method is presented. First, the basics of the SGPR used will be dealt with. Building on this, the entire concept of the procedure is described with a subsequent verification of the real-time capability.

A. Sparse Gaussian Process Regression

The inverse dynamics describes the relationship between the torques and the corresponding joint angles and its derivatives which has to be learned. Hence, with an independent consideration of the joints $j = 1, ..., \dim(\boldsymbol{\tau})$ the following applies

$$\tau_j = \mathcal{ID}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}), \tag{1}$$

or more generally

$$f : \boldsymbol{x} \mapsto y \tag{2}$$

with inputs $\boldsymbol{x} \in \mathbb{R}^{\dim(\boldsymbol{q}^T, \dot{\boldsymbol{q}}^T, \ddot{\boldsymbol{q}}^T)}$ and output $y \in \mathbb{R}$. The noise in the system $\epsilon_i \sim \mathcal{N}(0, \sigma^2_{\text{noise}})$ is assumed to be Gaussian with zero-mean and variance σ^2_{noise}, so that finally the relation

$$y_i = f_i + \epsilon_i \tag{3}$$

applies to a data point i. Here, $f_i = f(\boldsymbol{x}_i)$ describes the model to be learned which - with the additive noise component - is equal to the output variable y_i.

For a known training set $\boldsymbol{\mathcal{D}} = \{(\boldsymbol{x}_i, y_i), i = 1, ..., n\}$ with n data points, there is a regression problem with the aim to predict f^* at a query point $\boldsymbol{x}^*$. The simpler interpretation of the model compared to neural networks as well as the determination of uncertainties in contrast to other kernel machines are two advantages for a Gaussian Process Regression.

By definition, a GP is a set of random variables f_i for which each finite subset follows a Gaussian distribution. Since the GPR is a Bayesian approach, we have to define the prior distribution. We assume zero-mean which seems plausible because we can normalize the outputs to zero-mean. In terms of covariance, we define $\boldsymbol{K}$ as the kernel matrix with the entries $K_{\boldsymbol{x}\boldsymbol{x}'} = \text{kernel}(\boldsymbol{x}, \boldsymbol{x}')$ and initially any kernel function. Further, $\boldsymbol{X} = (\boldsymbol{x}_1, ..., \boldsymbol{x}_n)^T$ describes the inputs and $\boldsymbol{y} = (y_1, ..., y_n)^T$ the related noisy observations. For the sake of clarity, we will refrain from writing out the conditioning on the inputs $\boldsymbol{X}, \boldsymbol{x}^*$ in the following explanations.

Using Bayes' rule we obtain the joint posterior distribution by considering the data. After marginalizing out the unwanted latent function values from the training set, we get the desired Gaussian predictive distribution [11]. The process of learning takes place by adapting the hyperparameters $\{\sigma^2_{\text{noise}}, \boldsymbol{\theta}\}$ to the data set where $\boldsymbol{\theta}$ stands for the hyperparameters of the selected kernel function. This process can be carried out by maximizing the marginal likelihood $p(\boldsymbol{y})$. So for each query point $\boldsymbol{x}^*$, we can make a prediction for f^* consisting of mean and variance. However, the problem is the size of the kernel matrix of the inputs $\boldsymbol{K}_{\boldsymbol{XX}}$. Due to the very computational inversion of a n x n matrix, the training time scales with $\mathcal{O}(n^3)$ and the prediction time for a test point with $\mathcal{O}(n)$. Many data points n therefore require a lot of computation. This property is not acceptable for an online approach in which the models are continuously adapted to new data.

This problem can be solved by using a computationally efficient sparse method. Thereby, the general approach is the execution of complex arithmetic operations on a set of m data points which is much smaller than the full set. An overview of possible sparse approximations has already been given [11].

In this work, we use the Deterministic Training Conditional Approximation [12]. The likelihood is approximated with

$$q(\boldsymbol{y}|\boldsymbol{f}_m) = \mathcal{N}((\boldsymbol{K}^{-1}_{mm}\boldsymbol{K}_{m\boldsymbol{X}})^T\boldsymbol{f}_m, \sigma^2_{\text{noise}}\boldsymbol{I}) \tag{4}$$

where $\boldsymbol{f}_m$ describes the latent funtion values of the reduced set and $\boldsymbol{I}$ the identity matrix. The subscript m of the kernel matrices represents $\boldsymbol{X}_m$ which is the matrix with m inputs. Thus, the resulting predictive distribution at $\boldsymbol{x}^*$ consisting of mean value μ and variance σ^2 can be calculated [12].

Since the likelihood approximation also causes an approximation of the marginal likelihood, the expression

$$q(\boldsymbol{y}) = \mathcal{N}(\boldsymbol{0}, \boldsymbol{K}_{\boldsymbol{X}m}\boldsymbol{K}_{mm}^{-1}\boldsymbol{K}_{\boldsymbol{X}m}^T + \sigma_{\text{noise}}^2\boldsymbol{I}) \tag{5}$$

now has to be maximized in order to optimize the hyperparameters. Note that in our approach the inducing inputs $\boldsymbol{X}_m$ are variational which means that they play the role of extra kernel hyperparameters. Thus, the parameters $(\boldsymbol{X}_m, \boldsymbol{\theta}, \sigma_{\text{noise}}^2)$ have to be optimized.

In summary, it can be said that with such an approach only m x m matrices have to be inverted which significantly reduces the computational effort depending on the number of inducing inputs m considered. This means that both the training and the prediction can be executed faster which is essential for the online learning of inverse dynamics and the calculation of the required feedforward moments.

B. Concept for Feedforward Control

After the regression method used was introduced, the entire concept of pre-control can now be presented. This is illustrated in Fig. 1. As input, there is a new data point $\{\boldsymbol{x}_t, \boldsymbol{y}_t\}$ at the time step t as well as the input variables for the next time step $\boldsymbol{x}_{t+1}$. For the latter, the method should approximate the corresponding output quantities. Within the feedforward control, the models are updated with the current state of the robot in order to predict the moments in the next time step by means of desired joint sizes.

As already described, the first step is to update the n_{GP} models. You can freely choose how many GP models drift parallel along the data. Each GP k has the set ${}_k\boldsymbol{\mathcal{D}}$ with a different number of data points. The new point is inserted in this set. Due to the use of a database $\boldsymbol{\mathcal{DB}}$ to store old data points, the oldest data point of the GP model with the index $k_{\min}$ is then added to it. This GP contains the fewest points from the past, so when storing the oldest data point from this model, it can be ensured that the database can provide all data points from the past for each GP. Since the size of the database for an online procedure should be limited due to the storage space required, it is therefore possible that the database is completely filled. In this case, the oldest data point should be deleted from the database to ensure a constant size. Also, as soon as the respective set ${}_k\boldsymbol{\mathcal{D}}$ is completely filled, the oldest data point is deleted from this GP.

After the storage, the training starts. First, the respective training set ${}_k\boldsymbol{\mathcal{D}}^{\text{Train}}$ is created which consists, on the one hand, of the data set of the GP. On the other hand, the database can optionally be used by inserting the knn-next neighbors ${}_{knn}\boldsymbol{\mathcal{DB}}$ from the inputs in the next time step $\boldsymbol{x}_{t+1}$ from the database into the training set. Thus, there are many data points in the neighborhood of the query point which results in a higher accuracy of the prediction. With the library scikit-learn [13] such a search of the nearest neighbors can be performed.

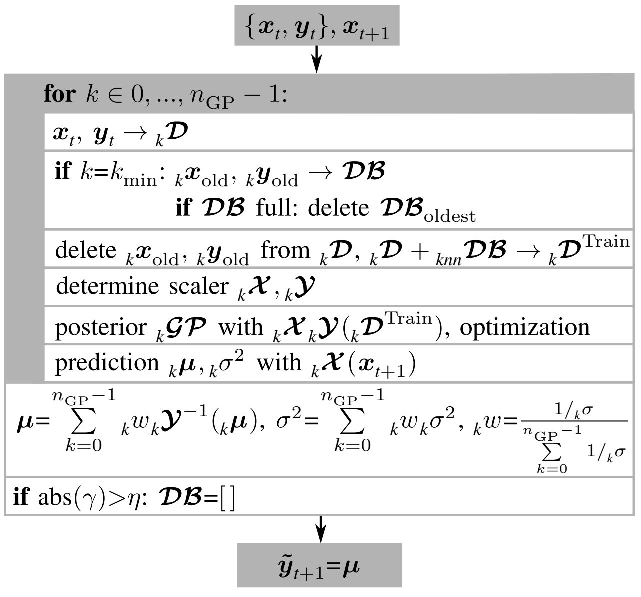
$\{\boldsymbol{x}_t, \boldsymbol{y}_t\}, \boldsymbol{x}_{t+1}$
for $k \in 0, ..., n_{\text{GP}} - 1$:
$\boldsymbol{x}_t,\ \boldsymbol{y}_t \rightarrow {}_k\boldsymbol{\mathcal{D}}$
if k=$k_{\min}$: ${}_k\boldsymbol{x}_{\text{old}}, {}_k\boldsymbol{y}_{\text{old}} \rightarrow \boldsymbol{\mathcal{DB}}$
if $\boldsymbol{\mathcal{DB}}$ full: delete $\boldsymbol{\mathcal{DB}}_{\text{oldest}}$
delete ${}_k\boldsymbol{x}_{\text{old}}, {}_k\boldsymbol{y}_{\text{old}}$ from ${}_k\boldsymbol{\mathcal{D}}$, ${}_k\boldsymbol{\mathcal{D}} + {}_{knn}\boldsymbol{\mathcal{DB}} \rightarrow {}_k\boldsymbol{\mathcal{D}}^{\text{Train}}$
determine scaler ${}_k\boldsymbol{\mathcal{X}}, {}_k\boldsymbol{\mathcal{Y}}$
posterior ${}_k\boldsymbol{\mathcal{GP}}$ with ${}_k\boldsymbol{\mathcal{X}}\,{}_k\boldsymbol{\mathcal{Y}}({}_k\boldsymbol{\mathcal{D}}^{\text{Train}})$, optimization
prediction ${}_k\boldsymbol{\mu}, {}_k\sigma^2$ with ${}_k\boldsymbol{\mathcal{X}}(\boldsymbol{x}_{t+1})$
$\boldsymbol{\mu}=\sum_{k=0}^{n_{\text{GP}}-1} {}_kw\,{}_k\boldsymbol{\mathcal{Y}}^{-1}({}_k\boldsymbol{\mu}),\ \sigma^2=\sum_{k=0}^{n_{\text{GP}}-1} {}_kw\,{}_k\sigma^2,\ {}_kw=\frac{1/{}_k\sigma}{\sum_{k=0}^{n_{\text{GP}}-1} 1/{}_k\sigma}$
if abs$(\gamma)>\eta$: $\boldsymbol{\mathcal{DB}}$=[ ]
$\tilde{\boldsymbol{y}}_{t+1}$=$\boldsymbol{\mu}$

Fig. 1. For each incoming data point $\{\boldsymbol{x}_t,\ \boldsymbol{y}_t\}$ the parallel drifting GP models are updated first and then used to make a prediction of the outputs for the known input variables in the next time step $\boldsymbol{x}_{t+1}$.

Due to the considerable differences in magnitude between the individual quantities, e.g. between joint angles and joint accelerations, the data must be preprocessed. There, we scale the inputs as well as the outputs so that they are zero-mean and have a unit variance. The scikit-learn library can also be used for this purpose. With this, the scaler ${}_k\boldsymbol{\mathcal{X}}, {}_k\boldsymbol{\mathcal{Y}}$ are determined with the inputs and outputs of the data set ${}_k\boldsymbol{\mathcal{D}}$. Then the processed data ${}_k\boldsymbol{\mathcal{X}}\,{}_k\boldsymbol{\mathcal{Y}}({}_k\boldsymbol{\mathcal{D}}^{\text{Train}})$ can be used to determine the posterior ${}_k\boldsymbol{\mathcal{GP}}$ whose hyperparameters are optimized in a next step. With such an optimized model we can finally perform the desired prediction. Note, that also here the query point $\boldsymbol{x}_{t+1}$ must be transformed with the scaler ${}_k\boldsymbol{\mathcal{X}}$. Accordingly, the output variables, i.e. the expected value ${}_k\boldsymbol{\mu}$ and the corresponding scalar variance ${}_k\sigma^2$, are also scaled. For the SGPR, we use the library GPy [14]. With this, a regression with multidimensional outputs is possible, which is why we also use outputs of the dimension dim($\boldsymbol{\tau}$) in the notation. In detail, an independent treatment of the individual outputs is carried out here. Therefore, the statistical methods used correspond to those described in the section II-A.

After this procedure has been performed for all GPs, the next step is the final approximation $\tilde{\boldsymbol{y}}_{t+1} = \boldsymbol{\mu}$ for the unknown outputs in the next time step. The respective predictions of the individual models ${}_k\boldsymbol{\mu}$ are weighted with ${}_kw$. Because the expected values are still scaled, they first have to be transformed back with the scaler ${}_k\boldsymbol{\mathcal{Y}}$. When determining the weighting factors, we use the reciprocal of the standard deviation. With a more accurate prediction, the standard deviation is smaller than with a poor approximation. Consequently, predictions with lower standard deviations should be weighted higher resulting in reciprocal value formation. Similarly, the variance of the final prediction can be obtained by weighting the variances of each model. Since these are scaled, the variance of the final prediction σ^2 is also scaled.

In a final step, a possible change in the system dynamics is taken into account. If the system behavior changes for unknown reasons, all data points collected so far would no longer match the system. The added knn nearest neighbors from the database would belong to an outdated system. For this reason, the database is completely deleted if the magnitude of an error measure γ to be defined exceeds a threshold η. Thus, a change in the system dynamics is assumed if the approximation is too inaccurate.

C. Verification of Real-Time Capability

After a detailed description of the feedforward concept based on online learning in the previous section, it is necessary to answer a central question: Is this procedure real-time capable? In order to answer this, in the sense of the definition of real-time capability mentioned above, it must be ensured whether the method reliably delivers a prediction $\tilde{\boldsymbol{y}}_{t+1}$ for each control cycle. For a controller, the defined cycle time should therefore be an upper limit for the computing time required for a run. For this, we define a typical cycle time of 5 ms.

For verification, experiment data of an industrial robot are used with $\dim(\boldsymbol{q}^T, \dot{\boldsymbol{q}}^T, \ddot{\boldsymbol{q}}^T) = 9$ and $\dim(\boldsymbol{\tau}) = 3$. In addition, two GP models drift along the data of size $n_1 = 40$ and $n_2 = 80$, which are extended with additional $knn = 20$ database points. The number of inducing inputs for the SGPR is $m = 20$ and the known RBF kernel is used as kernel function

$$\text{kernel}(\boldsymbol{x}, \boldsymbol{x}') = \sigma^2_{\text{rbf}} \exp(-\frac{1}{2l^2}||\boldsymbol{x} - \boldsymbol{x}'||^2) \tag{6}$$

with the RBF variance σ^2_{rbf} and the lengthscale l as hyperparameters. With the inducing inputs $\boldsymbol{X}_m$ and the noise variance σ^2_{noise} there are finally 183 hyperparameters which have to be optimized. To ensure convergence, the optimization is carried out with a maximum of ten iterations in order to keep the gradients of the parameters satisfactorily low.

The result of the investigation is that the run shown in Fig. 1 takes about 90 ms. To speed up, we change the algorithm so that in each run only one model is updated. Consequently, both the creation of the posterior with the new data and the optimization only have to be performed once per run. These two steps cost the most time. With a change like this, the computing time can be halved. However, the required 45 ms are still significantly above the cycle time which is not acceptable for an implementation. Therefore, the described procedure is not real-time capable in its form. Incidentally, even if the cycle time had been observed, the Python environment used on a Windows operating system would by definition not be real-time capable. The adjustments we made to implement the chosen approach in a robot controller are presented in the following section.

III. Implementation in the Robot Controller

This chapter describes in detail how we proceeded with the implementation. The main task here is to change the procedure illustrated in Fig. 1 in such a way that there are torques for feedforward control in each cycle available. An intuitive approach would be to accelerate the process so that the time for one run is less than the defined cycle time of 5 ms. This could be achieved for example by reducing the number of inducing inputs m or the number of optimization iterations. Disadvantage of such changes is the loss of accuracy of all GP models and, consequently, of all approximations. This is the reason why we are pursuing a different philosophy: splitting the procedure into two threads running in parallel. Thereby, we separate the prediction process from the computing-intensive training which results in the architecture shown in Fig. 2.

Accordingly, the developed program PGPOL (Parallel Gaussian Processes for Online Learning) consists of two threads, the *model update* and the *prediction*. Since PGPOL runs on a Windows operating system, communication with the Programmable Logic Controller (PLC) for pre-control must take place. This exchange is realized by an UDP-IP connection whereby each thread uses a different port.

Starting with the model update: analogous to Fig. 1, this thread receives a current data point consisting of actual joint sizes $\boldsymbol{x}_{t-5}$ and actual moments $\boldsymbol{y}_{t-5}$ of the robot as input. Due to the bus runtime, this data point effectively belongs to the time step $t-5$. With this, all parallel drifting GP models can be updated according to the known scheme. When using the database, the knn-next neighbors ${}_{knn}\boldsymbol{\mathcal{DB}}$ of desired joint sizes in two time steps $\boldsymbol{x}_{\text{d},t+2}$ are additionally inserted into the respective training set. When updating only one GP per run as described above, this process takes 45 ms, with the sizes specified in section II-C including the UDP-IP communication.

Parallel to this, a permanent prediction loop is executed. This thread receives the error measure

$$\gamma = \overline{\text{nmse}}_{t-5} - \overline{\text{nmse}}_{t-6} \tag{7}$$

which is the change of the normalized mean squared error averaged over all joint axes j

$$\overline{\text{nmse}}_t = \sum_{j=1}^{\dim(\boldsymbol{\tau})} \frac{\text{nmse}_{t,j}}{\dim(\boldsymbol{\tau})}. \tag{8}$$

The used normalized mean squared error from axis j at time step t is calculated by

$$\text{nmse}_{t,j} = \frac{\frac{1}{t}\sum_{i=1}^{t}(y_{i,j} - \tilde{y}_{i,j})^2}{\text{var}(\boldsymbol{y}_j)}. \tag{9}$$

In the numerator, there is the mean squared error between the actual moment of the respective axis $y_{i,j}$ and the corresponding approximated moment $\tilde{y}_{i,j}$ along the entire time steps i=1,...,t. This is normalized by the variance of the actual moments $\boldsymbol{y}_j$ of the axis j up to the time step t. The calculation takes place incrementally on the PLC at each time step. Note that due to

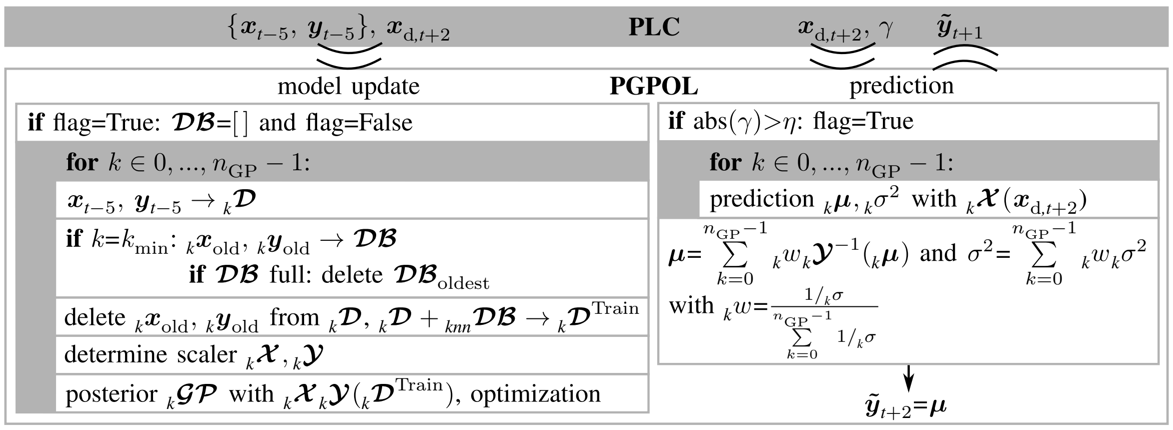
$\{\boldsymbol{x}_{t-5}, \boldsymbol{y}_{t-5}\}, \boldsymbol{x}_{\mathrm{d},t+2}$
PLC
$\boldsymbol{x}_{\mathrm{d},t+2}, \gamma$
$\tilde{\boldsymbol{y}}_{t+1}$
model update
PGPOL
prediction
if flag=True: $\boldsymbol{\mathcal{DB}}$=[ ] and flag=False
for $k \in 0, ..., n_{\mathrm{GP}}-1$:
$\boldsymbol{x}_{t-5}, \boldsymbol{y}_{t-5} \rightarrow {}_k\boldsymbol{\mathcal{D}}$
if k=k_{min}: ${}_k\boldsymbol{x}_{\mathrm{old}}, {}_k\boldsymbol{y}_{\mathrm{old}} \rightarrow \boldsymbol{\mathcal{DB}}$
if $\boldsymbol{\mathcal{DB}}$ full: delete $\boldsymbol{\mathcal{DB}}_{\mathrm{oldest}}$
delete ${}_k\boldsymbol{x}_{\mathrm{old}}, {}_k\boldsymbol{y}_{\mathrm{old}}$ from ${}_k\boldsymbol{\mathcal{D}}$, ${}_k\boldsymbol{\mathcal{D}} + {}_{knn}\boldsymbol{\mathcal{DB}} \rightarrow {}_k\boldsymbol{\mathcal{D}}^{\mathrm{Train}}$
determine scaler ${}_k\boldsymbol{\mathcal{X}}, {}_k\boldsymbol{\mathcal{Y}}$
posterior ${}_k\boldsymbol{\mathcal{GP}}$ with ${}_k\boldsymbol{\mathcal{X}}\,{}_k\boldsymbol{\mathcal{Y}}({}_k\boldsymbol{\mathcal{D}}^{\mathrm{Train}})$, optimization
if abs(γ)>η: flag=True
for $k \in 0, ..., n_{\mathrm{GP}}-1$:
prediction ${}_k\boldsymbol{\mu}, {}_k\sigma^2$ with ${}_k\boldsymbol{\mathcal{X}}(\boldsymbol{x}_{\mathrm{d},t+2})$
$\boldsymbol{\mu}=\sum_{k=0}^{n_{\mathrm{GP}}-1} {}_kw\,{}_k\boldsymbol{\mathcal{Y}}^{-1}({}_k\boldsymbol{\mu})$ and $\sigma^2=\sum_{k=0}^{n_{\mathrm{GP}}-1} {}_kw\,{}_k\sigma^2$
with ${}_kw=\frac{1/{}_k\sigma}{\sum_{k=0}^{n_{\mathrm{GP}}-1} 1/{}_k\sigma}$
$\tilde{\boldsymbol{y}}_{t+2}=\boldsymbol{\mu}$

Fig. 2. The implementation of the approach presented is done by splitting the procedure into two threads. Parallel to the computation-intensive model update runs the prediction, for which the current GP models are used. An Internet Protocol enables data exchange between the program and the controller.

the already mentioned offset by the bus runtime, the defined measure γ describes the error change between the steps $t-5$ and $t-6$. At the beginning of each prediction run, the program first checks whether this error amount exceeds a threshold η. If this is the case, a change in the system dynamics is assumed. A flag is used to clear the database in the model update thread.

In addition, this thread includes prediction, which has already been explained in section II-B. Since our architecture is chosen in such a way that both sending and receiving take place at the beginning of each run, the calculation has to be time-shifted. This means that the moments approximated to the time step t are sent to the PLC in the following time step. Consequently, a prediction with the desired joint sizes $\boldsymbol{x}_{\mathrm{d},t+2}$ must be performed so that the approximation $\tilde{\boldsymbol{y}}_{t+2}$ can be saved. In the following time step, this vector can then be sent to the PLC. Hence, the controller has effectively approximated moments for the step $t+1$. These can then be pre-controlled in the correct time, i.e. in the next time step.

Finally, it must be noted that the required joint sizes $\boldsymbol{x}_{\mathrm{d},t+2}$ are not available on the controller due to the software. Only the desired sizes for the next time step are calculated. In order to solve this problem, we have decided on a simple linear extrapolation which approximates the required joint sizes with existing desired values at time step t and $t+1$.

By splitting the approach, it is now possible that updated feedforward moments are theoretically available in each time step. This can be realized because the cycle time of the prediction thread including communication is on average less than the specified cycle time of 5 ms. Be aware that due to the non real-time environment, it is still possible that this limit may be exceeded. If there are therefore no feedforward moments, the previous values are then pre-controlled again.

IV. Experimental Validation

In order to validate the proposed approach for pre-control, experiments are carried out on a parallel kinematic robot. In detail, it is an industrial Codian D4-1100 delta robot designed for highly dynamic pick-and-place applications and controlled by a standard industrial PLC.

A. Movement along a Trajectory

To investigate the database approach and generally the use of parallel drifting GP models, a defined trajectory with the robot is driven in a first experiment. This consists of a square with a subsequent circle and is illustrated in Fig. 3a). The experiment is performed for different constellations of the GP models. Furthermore, an offline identified rigid body model of the delta robot including both the motor inertia and friction is used for feedforward control as comparison. The tracking error is examined more closely as a performance criterion, since the main goal of feedforward control is to reduce this error. The calculation of the squared error area of the joint j

$$A_j = \int_{\tau}^{\tau+T} e_j^2 \mathrm{d}t \tag{10}$$

is useful here where the tracking error of the respective axis e_j is squared and integrated over a defined duration T. Due to discrete values, numerical integration is necessary, whereby the trapezoidal rule is used as the quadrature formula.

Furthermore, it must be noted that the tracking error curve behaves aperiodically mainly with pre-control by PGPOL. The reason for this is the continuous adaptation of the models to new data. Even if this non-periodic behavior only occurs to a small extent, it should nevertheless be taken into account in the evaluation. Because of this, the integration is not performed over the required time for one drive of the selected trajectory, but over a much longer range $T = 40$s. During this time, the trajectory is carried out by the robot sufficiently often so that the aperiodicity of the tracking error in the error area is also taken into account. By summation over all joints j the total squared error area results in

$$A_{\mathrm{total}} = \sum_{j=1}^{\dim(\boldsymbol{\tau})} A_j. \tag{11}$$

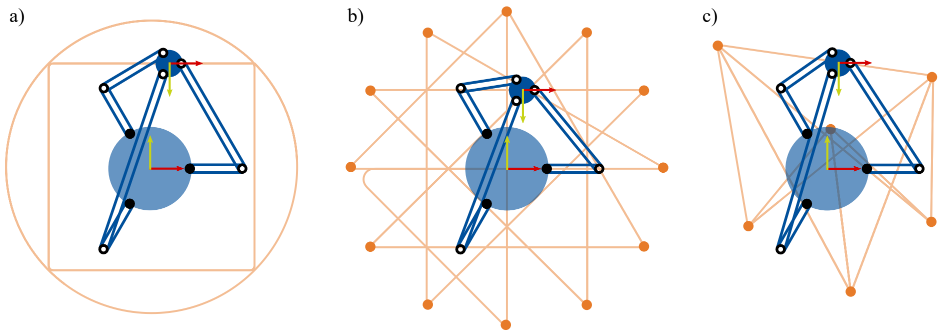

Fig. 3. Delta robot in top view with three different trajectories (orange): a) Movement along square with following circle b) Movement with many pick-and-place points (orange points) c) Movement with few pick-and-place points

Table I
TOTAL SQUARED AREA OF TRACKING ERRORS: EXPERIMENT I

approach to feedforward control	A_{total} in deg^{2}ms
without feedforward	426.2
offline identified model	213.7
$n_1 = 60$, $knn = 0 \rightarrow knn = 20$	$204.9 \rightarrow 203.4$
$n_1 = 120$, $knn = 0 \rightarrow knn = 20$	$204.5 \rightarrow 198.9$
$n_1 = 480$, $knn = 0 \rightarrow knn = 20$	$194.8 \rightarrow 199.2$
$n_1 = 60$, $n_2 = 120$, $knn = 0 \rightarrow knn = 20$	$203.9 \rightarrow 197.5$
$n_1 = 60$, $n_2 = 480$, $knn = 0 \rightarrow knn = 20$	$195.7 \rightarrow 194.6$

The results of the experiment are shown in Table I. First, compared to operation without any feedforward, the squared error area can be approximately halved by using the offline identified rigid body model for pre-control. The alternative approach, i.e. online learning of inverse dynamics using GPR, performs even better in any constellation. Thereby, it is noticeable that all the error areas are very similar in magnitude.

The best result can be obtained with the parallel approach with models of size $n_1 = 60$ respectively $n_2 = 480$ and $knn = 20$ additional database points. To illustrate the quality of this constellation and, thus, the applied feedforward control, the actual moments are compared with the pre-controlled torques in Fig. 4, exemplary shown for joint 3. Here, an excerpt of the entire experiment is considered in which the whole trajectory was once run through. It is noticeable that in the smoother regions, the predictions are very close to the actual values. By contrast, at the peaks in particular there are large deviations. The main reason for this is that the RBF kernel used is primarily suitable for smooth functions. The learning of rough regions which are present at the peaks is therefore not ideal with this kernel function.

With one exception, the use of the database results in an improvement because the error area will be reduced further. A clear increase of performance by using several parallel drifting Gaussian Processes, however, is not obvious by this experiment, since the error area for the best approach with a single GP is only minimally larger as the area of the best parallel constellation. It should be mentioned that these results refer only to the trajectory with defined speed settings. Thus, parallel drifting GPs can be useful for other movements.

For the following experiment, we therefore keep the parallel approach which performed best in this experiment. This constellation has reduced the total squared area of the tracking errors by more than 54%. Compared to the identified rigid body model, this is a clear improvement. At this point, it should be mentioned that the presented approach can be used in any serial or parallel kinematic robot. An elaborately generated robot model is put in the shade, at least in our experiment. In comparison, the constellation with a single GP of the size $n_1 = 480$ without using the database is additionally examined, since this enabled a similar performance. This mode is referred to as SGPOL (Single Gaussian Process for Online Learning) and the parallel mode is referred to as PGPOL.

B. Pick-and-Place Experiment

After we described the performance of our approach for the trajectory in the previous experiment, we will now investigate a possible change of the system dynamics. Thereby, we put the robot in a pick-and-place movement for which the parallel kinematic machine used is typically applied in practice. We simulate a change in system dynamics by increasing the speed of the end effector during operation. As a result, the joint moments are lifted to a different level. An additional mass at

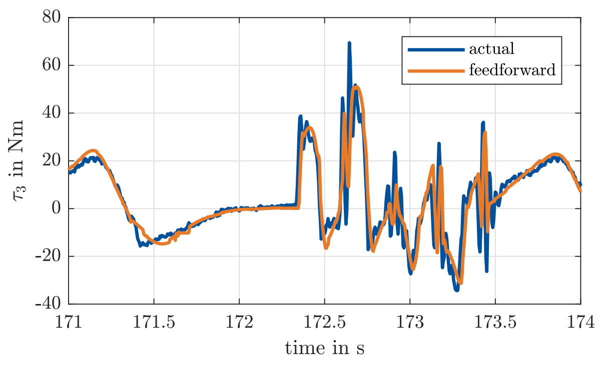

Fig. 4. Excerpt of the torque curve of axis 3 for the pre-control experiment with the best GP constellation: actual moments vs. feedforward moments

Table II
TOTAL SQUARED AREA OF TRACKING ERRORS: EXPERIMENT II

approach to feedforward control	A_{total} in deg^{2}ms
without feedforward	646.1
offline GP	320.9
PGPOL	322.9
SGPOL	311.7

the end effector would produce a similar effect. Therefore, we can use this to highlight the behavior to system changes.

As a comparison to the two described modes, i.e. SGPOL and PGPOL, we use an offline trained GP model. Hence, the first step is to learn the inverse dynamics offline with sufficient data. For this, we use as a training data set the trajectory shown in Fig. 3b) consisting of twelve pick-and-place points which are arranged in a star shape. At each of these points, the end effector moves along the vertical axis which is characteristic of a pick-and-place movement. The training takes place via SGPR on 15,000 data points and with 1,500 inducing inputs. The model of such a constellation needs on average 2 ms for a prediction of all three torsion moments. Consequently, with additional communication time of approx. 2 ms, the required run time is smaller than the cycle time. Since we increase the speed in the actual experiment this also has to be done during the generation of the training data. Thus, when the training trajectory is carried out, the speed is increased further up to a defined maximum. It should already be mentioned here that at this maximum speed, there is a highly dynamic movement.

The test trajectory should be different so that we can check how the offline model approximates for unknown data areas. Eventually, such an offline trained GP model should be universal and not only explicit for the training data. Hence, we use a test trajectory consisting of six pick-and-place points illustrated in Fig. 3c), whereby each is at a different position than the twelve points of the training movement.

When carrying out the experiment, three different speeds are set one after the other: 33%, 66% and 100% of the maximum speed. At each speed, the entire trajectory is driven five times. Thus, the test movement is performed a total of 15 times per experiment. As a performance criterion, the total squared area of the tracking errors is calculated again. Table II contains these for the mentioned feedforward approaches in comparison to carrying out the experiment without any pre-control. At first, it can be seen that the error area is more than halved by each pre-control approach. There, the largest reduction is achieved by using a single GP. The parallel learning reduces the error area slightly less, similar to using the offline trained GP model, whereas the latter approach is partly enhanced.

In order to clarify why the use of parallel drifting GPs reduces the tracking errors less, the approximation error will be investigated in more detail. We determine the course of the average normalized mean squared error $\overline{\text{nmse}}_t$ (8) as a function of time. The resulting curves for the three pre-control experiments are presented in Fig. 5. Basically, it is recognizable that at the beginning of each experiment, the error is very large. With the online approaches, this makes sense because depending on the constellation, the procedure takes a certain amount of time to learn the inverse dynamics. For the offline trained GP model, however, this behavior also exists although the model is already at the beginning of the experiment completely determined and no longer adapts to the data in the course of the experiment. The reason for this process is the fact that the end effector is in a rest position before starting the movement. This state was not learned with the offline trained GP. Consequently, the error is on a higher level which is balanced after a short time.

Further, it is noticeable that the approximation error increases with rising speed (①→② and ②→③). This can be justified by the fact that at low speeds, the torques are clearly smoother compared to highly dynamic motion. With the RBF kernel, such a smooth behavior can be learned enhanced in contrast to the fast pick-and-place process. Furthermore, with slow movements, the joint sizes also have a smoother course. Our applied linear extrapolation to determine the required desired joint sizes therefore generates a smaller error in this area than with faster movements. Accordingly, in highly dynamic processes, the moments are predicted for approximated desired values which deviate more clearly from the exact desired values. This also increases the approximation error.

A closer look at the courses reveals that after a certain initial phase, the approximation error for the rest of the experiment is smaller with PGPOL feedforward than with the offline trained GP. In addition, throughout the experiment the approximation error of the parallel approach is lower than when using a single GP. If one only considered these time histories, one would assume that PGPOL would best reduce the tracking error due to the prediction accuracy. Table II says the opposite.

For clarification, we consider the squared error areas in sections. Since the speed in area ① is very low, the tracking errors are also low. In this section, the error areas of all three pre-control approaches are similar in amount. Consequently, a consideration of the remaining areas is expedient. It is noticeable that, depending on the area, either the offline

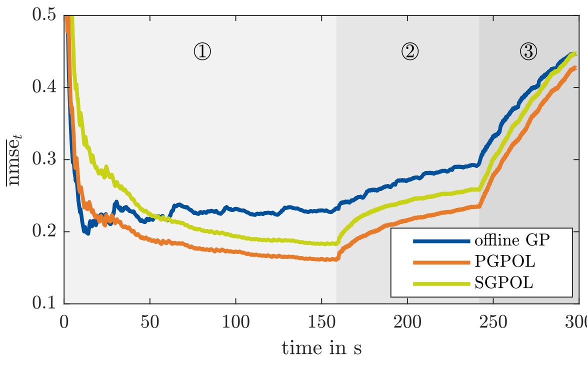

Fig. 5. Course of the approximation error for the pick-and-place experiment with feedforward by the offline GP, PGPOL and SGPOL. The three differently grayed sections represent the three respective speed settings.

approach or the online procedures have a smaller error area and are therefore better. In ②, the offline GP performs better than the online approaches. This behavior can be explained by the fact that with such a doubling of speed, the online approaches need time to adapt to the new dynamics. This is also substantiated by Fig. 5 because at the beginning of the region ②, the approximation error for the online procedures grows strongly. For longer drives in this area, it could be assumed that the online approaches would result in a more significant reduction of tracking errors than the offline GP due to the better prediction accuracy. In the area of maximum speed, the online approaches perform better. The speed change from ② to ③ is therefore not as significant as before. Hence, the online approaches can adapt faster.

Further, it is observed that the SGPOL mode reduces the tracking errors more than the use of parallel drifting GPs. This can also be justified by the adaptation to new data. Since the latter method consists of two models and only one GP is updated in each run of the model update thread, this approach takes longer to adapt. However, this is not visible in Fig. 5 because PGPOL performs better than SGPOL during the whole experiment. A pure consideration of the approximation error therefore does not necessarily provide information about the performance. Consequently, the tracking error reduction should also be investigated since this is the main task of pre-control.

V. Conclusions

This paper describes an approach for online learning using parallel drifting Gaussian Processes and a database extension. A central component is the implementation of such an approach in the robot controller in order to realize a feedforward control with the permanently adapted inverse dynamics.

Experiments show that any GP constellations perform better in terms of tracking error reduction than an identified physical model. In addition, it can be seen that using the database as a rule leads to an increase in the performance. Pick-and-place experiments with simulated dynamic changes show that parallel drifting GPs are advantageous regarding the approximation error. When considering the tracking error reduction, however, a single Gaussian Process in our investigations performs better. In general, the results give the impression that the presented use of several Gaussian Processes is not necessary and that a single GP provides comparable accuracy.

Despite all the positive results, new tasks arise from this work. It was found that in smoother regions, due to the property of the RBF kernel, the approximation is more accurate than in rough regions. Accordingly, a deeper investigation should be carried out with regard to the kernel choice. The architecture-related linear extrapolation of the desired values is also a source of errors. For the industrial application of such an approach, it is also desirable to ensure real-time capability so that current pre-control torques are reliably available in every cycle. Executing the process directly on the controller would represent the best solution as this eliminates the UDP-IP communication. The time required for this would be saved. In addition, the linear extrapolation required in our case would then no longer be necessary, since only desired values are required for the next time step which are provided by the software of the controller used.

Besides the method presented here, a task-specific approach would also be expedient. Depending on the task, specific models are trained. In relation to a pick-and-place robot, one could define one task as a motion with mass at the end effector and a second as a motion without mass. Data from sensors could thus provide information about which case is present and the respective Gaussian Process or the respective parallel drifting Gaussian Processes could be used. The chosen GP constellation would therefore not have to adapt to the new dynamics repeatedly after placing or picking up the mass, instead there would be a change in the model or models which is/are already trained for the respective case.

It is also possible to mix a model-based approach with the online learning method. If an identified inverse dynamics model of the robot is available, this existing knowledge could be used. The error of the identified model could be continuously learned online and additionally pre-controlled. A model identified offline can thus be improved by machine learning techniques.

References

[1] D. Nguyen-Tuong, J. Peters, M. Seeger, and B. Schölkopf, "Learning inverse dynamics: A comparison," in Proceedings of the European Symposium on Artificial Neural Networks, pp. 13–18, 2008.
[2] C. E. Rasmussen and C. K. I. Williams, Gaussian processes for machine learning, 3rd ed., ser. Adaptive computation and machine learning. 2008.
[3] B. Schölkopf and A. J. Smola, Learning with kernels: Support vector machines, regularization, optimization, and beyond, ser. Adaptive computation and machine learning. 2002.
[4] S. Vijayakumar and S. Schaal, "Locally weighted projection regression: An o(n) algorithm for incremental real time learning in high dimensional space," vol. 1, 2000.
[5] Y. Choi, S.-Y. Cheong, and N. Schweighofer, "Local online support vector regression for learning control," in International Symposium on Computational Intelligence in Robotics and Automation. pp. 13–18, 2007.
[6] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," Artificial Intelligence Review, vol. 11, no. 1/5, pp. 11–73, 1997.
[7] D. Nguyen-Tuong, M. Seeger, and J. Peters, "Local gaussian process regression for real time online model learning and control," in Advances in neural information processing systems 21, pp. 1193–1200, 2009.
[8] A. Gijsberts and G. Metta, "Real-time model learning using incremental sparse spectrum gaussian process regression," Neural Networks, vol. 41, pp. 59–69, 2013.
[9] F. Meier and S. Schaal, "Drifting gaussian processes with varying neighborhood sizes for online model learning," in IEEE International Conference on Robotics and Automation. pp. 264–269, 2016.
[10] D. Nguyen–Tuong and J. Peters, "Incremental sparsification for real-time online model learning," in Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, vol. 9, pp. 557–564, 2010.
[11] J. Quiñonero-Candela and C. E. Rasmussen, "A unifying view of sparse approximate gaussian process regression," in Journal of Machine Learning Research, vol. 6, pp. 1939–1959, 2005.
[12] M. Seeger, C. K. I. Williams, and N. Lawrence, "Fast forward selection to speed up sparse gaussian process regression," Artificial Intelligence and Statistics 9, 2003.
[13] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," in Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
[14] GPy, "Gpy: A gaussian process framework in python," http://github.com/SheffieldML/GPy, 2012.