PyGpPHs: A Python Package for Bayesian Modeling of Port-Hamiltonian Systems ⋆

Peilun Li * **Kaiyuan Tan** * **Thomas Beckers** *

* *Department of Computer Science, Vanderbilt University, Nashville, TN 37235, USA (e-mail: peilun.li@vanderbilt.edu, kaiyuan.tan@vanderbilt.edu, thomas.beckers@vanderbilt.edu)*

Abstract: PyGpPHs is a Python toolbox for physics-informed learning of physical systems. Compared to pure data-driven approaches, it relies on solid physics priors based on the Gaussian process port-Hamiltonian systems (GP-PHS) framework. This foundation guarantees that the learning procedure adheres to the fundamental physical laws governing real-world systems. Utilizing the Bayesian learning method, PyGpPHs enables physics-informed predictions with uncertainty quantification, which are based on the posterior distribution over Hamiltonians. The PyGpPHs toolbox is designed to make Bayesian learning with physics prior accessible to the learning and control community. PyGpPHs can be installed through an open-source link [1].

Keywords: port-Hamiltonian systems, physics-informed learning, Gaussian processes

1. INTRODUCTION

Modeling the dynamics of complex systems plays a pivotal role in making trustworthy and efficient control decisions. The applicability of such models extends across various domains, including but not limited to soft robots (Zheng and Lin, 2022), heat transfer processes (Zobeiry and Humfeld, 2021), and electrical systems (Cieślik, 2021). Whereas classical models of physical systems typically rely on first principles, there is a growing trend toward employing data-driven techniques to understand physical properties efficiently with reduced engineering effort. In response to the challenges presented by the complexity of these systems, various data-based methodologies have been proposed to address state forecasting tasks (Long et al., 2018; Bar-Sinai et al., 2019; Wu and Xiu, 2020; Stephany and Earls, 2022), among others. The fundamental concept underlying these methodologies is the formulation of system behavior derived from extensive observational data. Despite the advantages, these predominantly data-centric learning approaches raise pertinent concerns about the efficiency, safety, and especially the physical accuracy of the models they generate (Hou and Wang, 2013). A significant limitation arises from omitting physics-based priors in these models, making them potentially vulnerable and unreliable under specific conditions. Such inconsistencies are clearly untenable in applications demanding high accuracy and reliability. Consequently, there is an increasing interest among researchers in developing learning methodologies that are not only accurate, but also conform to the principles of physical reality. The concept of *physical reliability* is defined as the alignment of the learned model with established physical laws, ensuring its predictions roots on realities.

Integrating physics priors into the learning process poses the question of what framework can effectively encapsulate these physical priors. A substantial category of physical systems can be described through Hamiltonian mechanics, see (Nageshrao et al., 2015). One of the core principles of Hamiltonian mechanics is that the total energy of the system can be represented by a Hamiltonian function. The port-Hamiltonian system (PHS) framework is employed as extensions to Hamiltonian systems that can be used to model a large class of physical systems. This framework is particularly adept at encapsulating the essential components of physical systems, namely: 1) energy-storing elements, which are critical for maintaining the system's energy balance; 2) energy-dissipating or resistive elements, which account for energy loss within the system; and 3) energy-routing elements, which are pivotal in the distribution and transmission of energy throughout the system, see (Van Der Schaft and Jeltsema, 2014). The port-Hamiltonian framework thus serves as an instrumental tool in modeling physical priors, ensuring that the learning process is firmly anchored in the principles of physics. However, determining the Hamiltonian function for complex systems can be challenging. To address this, Gaussian process-port Hamiltonian systems (GP-PHS) has recently been introduced, see (Beckers et al., 2022), as a method to combine machine learning capabilities with physics-based priors. To bridge the gap between PHS theory and machine learning, we introduce this PyGpPHs toolbox for the community as a user-friendly way to use GP-PHS.

The contribution of this paper is the PyGpPHs toolbox, a physics-informed learning toolbox designed for simplicity in training and application. The proposed toolbox uses observed data of a system to learn the underlying Hamiltonian function with Gaussian processes. The toolbox offers an approach to performing Bayesian learning over all possible Hamiltonians without the prerequisite knowledge of its underlying optimized computations, which helps it

⋆ This work is supported by the Vanderbilt University Rapid-Advancement MicroGrant Program

[1] https://github.com/PyGpPhs-Source/PyGpPhs

strike a balance between computational efficiency and user accessibility.

The remainder of this paper is organized as follows. Section 2 introduces the framework for GP-PHS. The details of the `PyGpPHs` toolbox are presented in Section 3. Finally, the usability of the toolbox is demonstrated on a case study presented in Section 4.

2. THE GP-PHS FRAMEWORK

In this section, we recap GP-PHS, a recently proposed approach for physics-informed learning of dynamical systems (Beckers et al., 2022). In this approach, it is assumed that the system to be learned can be described as a port-Hamiltonian system [1]

$$\begin{aligned}\dot{\boldsymbol{x}} &= [J(\boldsymbol{x}) - R(\boldsymbol{x})]\nabla_x H(\boldsymbol{x}) + G(\boldsymbol{x})\boldsymbol{u} \\ \boldsymbol{y} &= G(\boldsymbol{x})^T \nabla_x H(\boldsymbol{x}),\end{aligned} \tag{1}$$

where, however, the Hamiltonian is unknown to us due to the system complexity. The idea of GP-PHS is leveraging state measurements to learn the unknown system Hamiltonian with a non-parametric Gaussian process (GP). Furthermore, the GP allows us to estimate any unknown parameters of the interconnection matrix J and the dissipation matrix R of the system by treating them as hyperparameters. This approach is beneficial in many ways: First, it does not rely on predetermined knowledge about the parametric structure of the Hamiltonian, which makes the proposed framework in particular suitable for highly nonlinear systems. Second, the Bayesian framework empowers the model to encapsulate all possible PHS configurations under the Bayesian prior, based on a finite set of observations. This aspect is not only crucial from the perspective of model identification, but can also play a vital role for potential robust model-based control approaches through its uncertainty quantification. Before introducing the details of the GP-PHS framework, we briefly introduce GPs, as they play a significant role in the model.

A Gaussian process is defined by its mean function m and the covariance function k, expressed as $\mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$. The kernel $k(\mathbf{x}, \mathbf{x}')$ determines the covariance between any two inputs, making GPs popular for non-parametric Bayesian regression. Given the input dataset $X = [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}]$ with outputs $Y = [y^{(1)}, \dots, y^{(N)}]$ as training set $\mathcal{D}$ sampled from an unknown function $y = f(\mathbf{x})$, a GP prior on f allows using Bayesian theorem to infer the posterior for a new input $\mathbf{x}^*$, updating the mean and covariance of the GP as following equations

$$\begin{aligned}\mu(y^*|\mathbf{x}^*) &= k(\mathbf{x}^*, X)K(X, X)^{-1}Y \\ \Sigma(y^*|\mathbf{x}^*) &= k(\mathbf{x}^*, \mathbf{x}^*) - k(\mathbf{x}^*, X)K(X, X)^{-1}k(\mathbf{x}^*, X),\end{aligned} \tag{2}$$

where $k(\mathbf{x}^*, X)$ represents the covariance between the new point and the training set, and $k(\mathbf{x}^*, \mathbf{x}^*)$ is the prior variance at the new point. Hence, the desired mean value and uncertainty measurement can be acquired by the learned predictive distribution $p(\mathbf{y}^*|\mathbf{x}^*, X, Y)$. See (Rasmussen and Williams, 2006) for more details on GPs.

[1] Vectors $\boldsymbol{a}$ and vector-valued functions $\boldsymbol{f}(\cdot)$ are denoted with bold characters. Matrices are described with capital letters. The matrix I_n is the n-dimensional identity matrix. The expression $A_{:,i}$ denotes the i-th column of A. The symbol $\mathbb{R}_{>0}$ denotes the set of positive real numbers, while $\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers. The operator $\nabla_{\boldsymbol{x}}$ with $\boldsymbol{x} \in \mathbb{R}^n$ denotes $[\frac{\partial}{\partial x_1}, \dots, \frac{\partial}{\partial x_n}]^\top$.

The challenges of employing GPs to learn the Hamiltonian are addressed in the following section, where we delineate the general framework of GP-PHS. This is followed by an in-depth discussion of the training and prediction processes associated with this modeling approach. For more details on GP-PHS, we refer the interested reader to (Beckers et al., 2022).

2.1 GP-PHS modeling

First, we start with the underlying assumptions.

Assumption 1. We assume that we have access to state observations of the system state $\boldsymbol{x}(t) \in \mathbb{R}^n$, whose evolution over time can be described as

$$\dot{\boldsymbol{x}} = [J(\boldsymbol{x}) - R(\boldsymbol{x})]\nabla_x H(\boldsymbol{x}) + G(\boldsymbol{x})\boldsymbol{u}. \tag{3}$$

Assumption 2. The system matrices J, R, and G are (partially) known, with the exception of a finite set of unknown parameters $\boldsymbol{\varphi}_J$, $\boldsymbol{\varphi}_R$, and $\boldsymbol{\varphi}_G$, respectively.

The GP-PHS model utilizes a GP prior on the estimated Hamiltonian $\hat{H} \sim \mathcal{GP}(0, k(\boldsymbol{x}, \boldsymbol{x}'))$ with a squared exponential kernel k, ensuring the smoothness of the Hamiltonian, and enables to learn any continuous function arbitrarily exactly due to its universality (Rasmussen and Williams, 2006). We exploit the GP's closure under affine transformations to integrate the PHS structure (3) into a GP. Hence, the GP-PHS evolves into $\dot{\boldsymbol{x}} \sim \mathcal{GP}(G(\boldsymbol{x})\boldsymbol{u}, k_{\text{phs}}(\boldsymbol{x}, \boldsymbol{x}'))$ with the PHS kernel

$$k_{\text{phs}}(\boldsymbol{x}, \boldsymbol{x}') = \sigma_f^2 J_R(\boldsymbol{x} \mid \boldsymbol{\varphi}_J, \boldsymbol{\varphi}_R)\Pi(\boldsymbol{x}, \boldsymbol{x}')J_R^\top(\boldsymbol{x}' \mid \boldsymbol{\varphi}_J, \boldsymbol{\varphi}_R),$$

where Π represents the Hessian of the squared exponential kernel. Further, $J_R(\boldsymbol{x} \mid \boldsymbol{\varphi}_J, \boldsymbol{\varphi}_R)$ denotes $J(\boldsymbol{x} \mid \boldsymbol{\varphi}_J) - R(\boldsymbol{x} \mid \boldsymbol{\varphi}_R)$ that represent the matrices J and R in the PHS (3) parameterized by the vectors of unknown parameters $\boldsymbol{\varphi}_J, \boldsymbol{\varphi}_R$. This kernel establishes a prior over port-Hamiltonian systems, allowing for posterior computation from training data to model the systems physically correct.

2.2 Training

In accordance with Assumption 1, we examine collected state observations as $\mathcal{D} = \{t_i, \hat{\boldsymbol{x}}(t_i), \boldsymbol{u}(t_i)\}_{i=1}^N$, where $\hat{\boldsymbol{x}}(t_i)$ denotes the state observed at time t_i, over a total of N time steps. This is related to the corresponding input series $\boldsymbol{u}(t_i)$. Given these observations $\mathcal{D}$, we aim to derive data in the form of $\{\boldsymbol{x}, \dot{\boldsymbol{x}}\}$, which allows us to apply GP regression with the PHS kernel to model the system dynamics. To calculate the derivative $\dot{\boldsymbol{x}}$, we again make use of the property that GPs are invariant under affine transformations. Using a differentiable kernel k, we train n distinct GPs, each for a separate dimension j of the state $\boldsymbol{x}$. These GPs are trained on subsets $\mathcal{E}_j = \{t_i, \hat{x}_j(t_i)\}_{i=1}^N$ for $j = 1, 2, \dots, n$. Consequently, we can create a new dataset $\dot{X}$ of estimated state derivatives using the mean prediction of each GP $\mu(\dot{x}_j \mid t, \mathcal{E}_j)$. The observed states $X = [\boldsymbol{x}(t_1), \dots, \boldsymbol{x}(t_N)]$ and the estimated state derivatives $\dot{X} = [\mu(\dot{\boldsymbol{x}}(t_1) \mid \mathcal{E}), \dots, \mu(\dot{\boldsymbol{x}}(t_N) \mid \mathcal{E})]$ allow us to learn the Hamiltonian H using the k_{phs} kernel and standard GP regression (2). Any unknown parameters in the kernel function, i.e., the unknown parameters in the J, R and G matrix, can be treated as hyperparameters and optimized by means of the log marginal likelihood function.

2.3 Prediction

In pursuit of generating predictions from the posterior distribution, we employ the joint distribution

$$\begin{bmatrix} \dot{X} \\ \boldsymbol{f}(\boldsymbol{x}^*) \end{bmatrix} = \mathcal{N}\left(0, \begin{bmatrix} K_{\text{phs}}(X,X) & K_{\text{phs}}(X,\boldsymbol{x}^*) \\ K_{\text{phs}}(\boldsymbol{x}^*,X) & K_{\text{phs}}(\boldsymbol{x}^*,\boldsymbol{x}^*) \end{bmatrix}\right) \quad (4)$$

at a new state $\boldsymbol{x}^* \in \mathbb{R}^n$. This framework facilitates the procurement of a vector field $\boldsymbol{f}$, representing the right-hand side of the port-Hamiltonian system (3). From this distribution, one can sample a $\boldsymbol{f}$, using some approximation technique for computational efficiency, and subsequently solve the resulting equations with an Ordinary Differential Equation (ODE) solver.

However, it is important to note that we have no guarantee that this interpolation will yield a dynamics function maintaining a PHS structure and adhering to the principles of energy conservation and dissipation inherent in the system. To address this challenge, we propose sampling the estimated Hamiltonian H, instead of the flow field $\boldsymbol{f}$. By leveraging the linear transformation $H \to \hat{J}_R \nabla H$, we can then derive the flow field, which is guaranteed to be generated from a PHS structure.

3. THE PYGPPHS TOOLBOX

3.1 Toolbox overview

The `PyGpPHs` toolbox is developed using Object-Oriented Programming in Python and C++. The main structure of the toolbox is written in Python, while a few computationally expensive functions, such as the kernel and the Cholesky decomposition to compute the inverse of the kernel matrix in (2), are written in C++ for computational efficiency. While two languages are used to develop the `PyGpPHs` toolbox, the notions of encapsulation and abstraction are largely realized. The C++ data structures and function calls are handled within the toolbox, thus removing explicit overheads for users to consider the C++ internal workings or invoke function calls.

The `PyGpPHs` toolbox consists of two classes and a set of functions, but the user interface of the toolbox is similar to those of other popular machine learning toolboxes. The two classes in the toolbox are named `model`, and `hyp`. The `model` class is the core of the toolbox and the embodiment of GP-PHS framework. This will be the class that interacts with the user for a variety of problem settings. The `hyp` class is a set of parameters that will be used for internal computation. In particular, the `hyp` class contains the hyperparameters of the kernel functions and parameters specific to the matrices in PHS. Instances of class `hyp` are not meant to be accessed directly from users, but this option is left open if the user considers initializing the parameters for the Gaussian process or the port-Hamiltonian system. While the `PyGpPHs` toolbox is efficient and extensive, there are some limitations at this point that will be addressed in future work. First, the dimension of the state variable x is limited to at most 2 for efficient and accurate modeling. Second, the matrices J and R in the PHS (3) are assumed to be constant and the input signal $u(t)$ being a scalar function. In addition to address these constraints, future work will include not only the mean prediction but also uncertainty quantification. However, the current `PyGpPHs` toolbox is a first step to provide simple and efficient way for users to learn dynamical systems. The following section will explain the functionality and implementation of the `model` class.

3.2 Toolbox usage

The `model` performs the learning on the port-Hamiltonian systems. Public methods can be categorized into three types: Initialization, Optimization, and Results.

Model Initialization

The initialization of the parameters is the first step in the model setup. In the toolbox, the parameters are kept in `hyp` class, and are defined to be sn, sd, l (the hyperparameters of the kernel function) and JR_vec (the elements of the J and R matrices). The user has the option to initialize the parameters in the `hyp` instance:

```
hpr=Hyp(JR=[[jr11, jr12, jr13],to_optimize])
```

Since J is a skew-symmetric matrix and R is a positive semi-definite matrix in the diagonal form, only the upper triangular part of $J - R$ is sufficient to uniquely define the model. To initialize JR_vec, the user needs to provide a list of two 1-dimensional arrays. The first array, $[jr_{11}, jr_{12}, jr_{13}]$, is a flattened upper triangular array of the $J-R$ matrix, and the array "to_optimize" is a boolean array specifying whether the element is suppose to be optimized. In other words, if the "to_optimize" array has $True$ at index i, then the i-th element from the upper triangular array will be optimized. Without initialization, the values are set to $J = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ and $R = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$. After setting the parameters, the model initialization is offered in two ways. The first way assumes that the user already has access to the state derivatives by measuring or utilizing estimation techniques. Thus, the user can provide the state variable observations X, time derivative of state variable observations dX, and the vector of time stamps of the observations T. It must be ensured that for N training points, i.e., T is a $1 \times N$ numpy array of time stamps, X, dX are $n \times N$ numpy arrays whose columns are state variable $\boldsymbol{x}$ and derivative of state variable $\dot{\boldsymbol{x}}$ respectively. In other words, X is the matrix whose columns are the observed states $\hat{\boldsymbol{x}}(t_i)$ at time $t_i \in T$, and dX is the matrix whose columns are the observed state derivatives $\dot{\boldsymbol{x}}(t_i)$.

In the second way where the user cannot provide measurements of the state derivatives, the `Model` function will invoke the `set_default_dX()` function and automatically generates dX. As described in Section 2, Gaussian process regression is used to approximate the state as a function of time. Then dX is computed by automatic differentiation with respect to time. For this initialization approach, the users need to define $G(\boldsymbol{x})$ and $u(t)$. The input/output matrix $G(\boldsymbol{x})$ should be defined as a Python function of a single state variable $\boldsymbol{x}$. Similarly, the input signal $u(t)$ should be a Python function of scalar time t.

```
model = Model(T, X, dX=None, G=G, u=u)
```

More specifically, if we have state variable $\boldsymbol{x} \in \mathbb{R}^n$, the model will train n separate GP regressions for every dimension of the state vector $\boldsymbol{x}$. For each GP, the predictive mean is computed, and the derivative at each $t_i \in T$

is evaluated using (Gardner et al., 2018) and PyTorch functions.

Optimization of Parameters
To train the model, the model must first be initialized with T, X and optionally dX as described above. After the model initialization, the user can proceed to optimize the Gaussian process hyperparameters as well as port-Hamiltonian system parameters. This can be done in a void or non-void behavior.

```
model.train()
```

The `train()` function calls the `optimize_hyp()` method that directly optimizes the necessary parameters for both the Gaussian process and port-Hamiltonian system, and then updates those parameters to the model. Then, updated parameters are used to calculate the Cholesky decomposition of the kernel matrix for later functionality. The user can access the parameters of interest through the get-methods provided by the model interface. In addition to the void behavior, the model further offers the option to return a Python dictionary of the optimized parameters

```
param_dict = model.train()
```

accessible via methods of dictionary data structure.

Predictions
The mean prediction of the learned Hamiltonian function can be accessed through

```
H_prediction = model.pred_H(X_test)
```

where the array `X_test` has columns being values of the state variables of interest. Since the Hamiltonian function has a scalar output, `pred_H` will evaluate each state value in `X_test` and consequently return an array of scalar values representing the Hamiltonian at each state.

By exploiting the PHS structure, the estimated time derivative of the state variable $\dot{\boldsymbol{x}}$ can be calculated based on the dynamics (3). Given a state $\boldsymbol{x}$, an input matrix G, and an input u, the predicted state derivative is returned by calling the following function.

```
dx = model.pred_dx(x, t, G, u)
```

In order to predict the trajectory of the state $\boldsymbol{x}$ for an array of time points $T_{span} = [0, \dots, t_n]$, i.e,

$$\boldsymbol{x}(t_k) = \boldsymbol{x}_0 + \int_0^{t_k} [J - R]\nabla_x H(\boldsymbol{x}) + G(\boldsymbol{x})u \, dt \qquad (5)$$

for all $t_k \in T_{span}$, with initial condition $\boldsymbol{x}_0 = \boldsymbol{x}(0)$, input matrix G, and u, the user can call the method as:

```
x = model.pred_trajectory(Tspan, x0, G, u)
```

This function utilizes the numerical integration package from Scipy. The time span $Tspan$ for the simulation can be chosen arbitrarily. If the input function u is not provided, a zero input will be assumed by default, thus, simulating the system without external influences.

4. CASE STUDY OF PYGPPHS

To demonstrate the usage and effectiveness of the `PyGpPHs` toolbox, we test it on a non-harmonic oscillator system visualized in Fig.1. The system consists of two magnets, one of which is fixed while the other is attached to a linear spring and damper. The two magnets have the same pole facing each other, exerting magnetic repulsion against each other. For a formal description of the problem

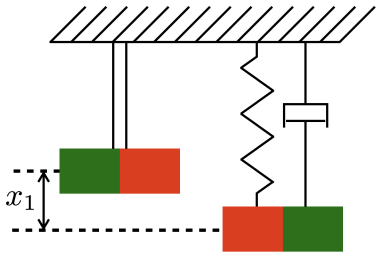

Fig. 1. Illustration of the magnet oscillator system.

setting, let $\boldsymbol{x} \in \mathbb{R}^2$ be the state variable of the port-Hamiltonian system, from which x_1 denotes the vertical position (m) of the magnet attached to the spring relative to the stationary magnet, and x_2 the momentum ($Kg \cdot m/s$) of the magnet. To evaluate the performance of the toolbox, we assume to know the underlying port-Hamiltonian dynamics of the system (3) with

$$\begin{aligned} H(\boldsymbol{x}) &= \frac{x_1^2}{2} + 2\cos(x_1) + \frac{x_2^2}{2} - 1 \\ G(\boldsymbol{x}) &= [0 \ 1]^\top, u(t) = 0.1\sin(t) \\ J - R &= \begin{bmatrix} 0 & 1 \\ -1 & -r \end{bmatrix}, \text{ where } r = 0.1. \end{aligned} \qquad (6)$$

as ground truth. We generate the training set by recording the state variable $\boldsymbol{x}(t)$ of the system (6) every $0.1s$ from $0s$ to $50s$ with initial state $\boldsymbol{x}_0 = [5, 0]^\top$. Hence, the vector of time stamps is $T \in \mathbb{R}^{501}$. The state observation $X = [\hat{\boldsymbol{x}}(t_1), \cdots, \hat{\boldsymbol{x}}(t_{501})]$, obtained from the simulation of the system (6), is of dimension $\mathbb{R}^{2\times 501}$. Furthermore, additive i.i.d. Gaussian noises are considered for the state observations $\hat{\boldsymbol{x}} \in X$. Fig. 2 shows the true system trajectory (blue) and the collected data points (red). After obtaining our training dataset, we initialize the model. First, the external Python libraries are imported accordingly, and the model initialization is achieved by the following code, given the training data T, X and the input matrix G and input $u(t)$ as defined in (6).

```
from PyGpPHs import Model
def G(x):
    return np.array([0, 1])
def u(t):
    return 0.1 * np.sin(t)
model = Model(T, X, dX=None, G=G, u=u)
```

We prepare the parameters for optimization as default, i.e., for the learning of the dynamics we assume an unknown damping factor in the R matrix of the system. During the initialization, the toolbox estimates the state derivatives

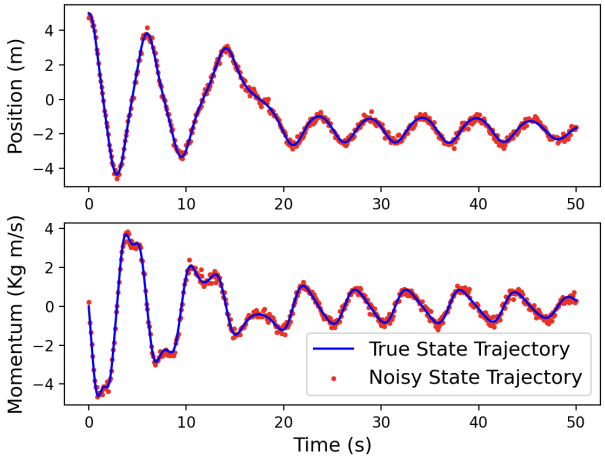

Fig. 2. The true system trajectory (blue) as ground truth and the noisy state observations (red) for the training of the GP-PHS model.

$dX = [\dot{\boldsymbol{x}}(t_1), \cdots, \dot{\boldsymbol{x}}(t_{501})]$ as we have not provided dX to the initialization function. The actual system dynamics is derived from (3) with parameters in (6), and we compare it with the estimation from the `PyGpPHs` toolbox. The accuracy of the estimation depends on the number and distribution of the training data.

After initializing the model, we may access the optimized Gaussian process hyperparameters and port-Hamiltonian system parameters either through *i*) `model` class accessors methods or *ii*) Python dictionary methods.

```
model.train()
JR = model.get_Hyp_JR()
```

The `train` function optimizes the necessary parameters for GP-PHS and updates those parameters to the model. Therefore, the user can access the parameters of interest through the get-methods provided by the model interface. Similarly, the model offers the option to return a dictionary of parameters.

```
param_dict = model.train()
JR = param_dict["JR"]
```

Now that the model is optimized, we examine the training of the model by comparing the learned Hamiltonian function (mean prediction) and the predicted trajectory with the ground truth. The Hamiltonian function describes the energy of the system given a state $\boldsymbol{x}$. To compare the learned Hamiltonian function with the true Hamiltonian, we define a grid of state variables.

```
tRange = np.arange(-4, 4.1, 0.1)
X1_test, X2_test = np.meshgrid(tRange, tRange)
x_test = np.vstack((X1_test.ravel(),
                    X2_test.ravel()))
H_pred = model.pred_H(x_test)
```

As the setting of the demonstration, variable $X1\_test$ denotes the vertical position of the springy magnet ranging

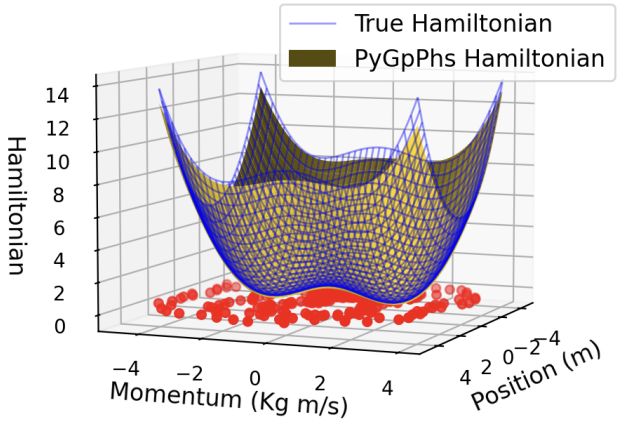

Fig. 3. The comparison between the true and learned Hamiltonian function based on collected data (red dots). The graph shows the accuracy of the learned Hamiltonian function.

from $-4m$ to $4m$, with interval to be $0.1m$, while the $X2\_test$ denotes the momentum of the magnet by a similar manner. Hence, combining both together gives us the set of points on which to get the Hamiltonian predictions, represented as red dots in Fig.3. After obtaining H_pred based on the learned Hamiltonian, its accuracy can be examined in comparison with the ground truth due to the known dynamics of the magnet system, shown in Fig.3.
In addition to the Hamiltonian function, the state trajectory $\boldsymbol{x}(t)$ can be predicted by exploiting the PHS structure in (3). The parameters for this specific magnet oscillator system, as well as the new initial condition $\boldsymbol{x}_0 = [-3, 0]^\top$ and a zero input $u(t) = 0$ are used to calculate the ground truth. We provide the same setting to the model by the following syntax.

```
u = lambda t : 0
x0 = np.array([-3, 0]).T
t_span = np.arange(0, 100.01, 0.01)
x = model.pred_trajectory(t_span, x0, G, u)
```

Fig.4 indicates that the model not only shows accurate results, but also follows physical principles due to the underlying PHS structure. When setting the initial condition to $\boldsymbol{x}_0 = [-3, 0]^T$, meaning the magnet is $3m$ lower than its counterpart, the system behavior changed accordingly. The initial condition is different from that of the training data, indicating that the model has not trained on or seen such dynamics. After oscillation, the magnet settled at a position different from that of the state observation from training. The PHS prior allows the model to generalize exceptionally well to previously unseen conditions as the new initial point in the example.

We further quantify the relation between accuracy and the number of state observations provided for training. Intuitively, a larger number of data can more comprehensively describe the system dynamics, thus better inform the learning process, and result in more accurate predictions. The Mean Squared Error (MSE) is used to measure the level of accuracy of the model prediction.

In the case study, the state observations $X \in \mathbb{R}^{2\times 501}$ are simulated from the true system with noise $\mathcal{N}(0, 0.15^2)$.

We evaluated by varying the number of state variables provided to the model. Specifically, if the number of provisions is k, the first k columns from X are provided to the model as the training data set. Hence, the new state observation $X_k \in \mathbb{R}^{2\times k}$, and $dX \in \mathbb{R}^{2\times k}$ is given accordingly. The MSE is then calculated as the average of five separate model executions. Fig.5 visualizes the relation between the number of data provided and the accuracy of the model. To reflect on the observation that the MSE does not monotonously decrease with larger amount of data provided, there are two factors contributing to the trend. First, the state observations are noisy by assumption, which could result in a higher MSE even if the number of training data is large. Furthermore, the parameters of the PHS, especially the damping factor in the R matrix, have a considerable influence on the quality of the prediction. Hence, if the model does not accurately learn the R matrix, this has an impact on the periodicity, duration, and magnitude of the oscillation. However, with more training data, the accuracy tends to increase in general.

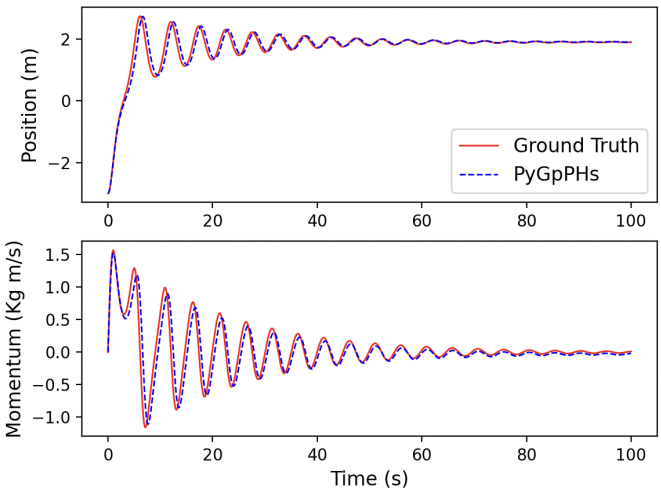

Fig. 4. The comparison between the ground truth trajectory (red) and the `PyGpPHs` predicted state trajectory with an unseen initial condition. The graph shows high accuracy and that the model generalizes well.

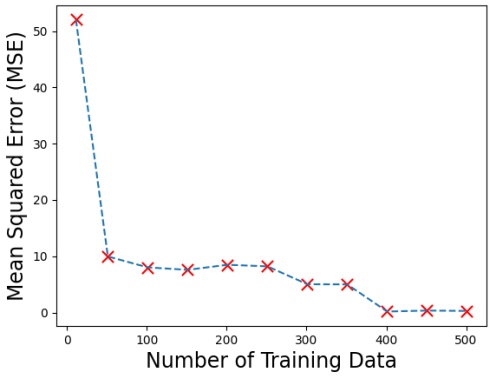

Fig. 5. The mean squared error (MSE) as a function of number of training data. The declining trends indicate more data usually leads to a more accurate model.

5. CONCLUSION

In conclusion, the `PyGpPHs` toolbox provides simple usability and efficient algorithms to use Gaussian process Port-Hamiltonian systems as a Bayesian learning approach for physical systems. The magnet oscillator case study demonstrates the simple user interface in line with the theoretical foundation as well as the accuracy and the generalizability of the model. `PyGpPHs` offers a reliable framework to incorporate foundational knowledge of physics into the domain of data-driven modeling of dynamical systems. Future work will include further optimization of performance, including systems with higher dimensions, and providing access to uncertainty quantification.

REFERENCES

Bar-Sinai, Y., Hoyer, S., Hickey, J., and Brenner, M.P. (2019). Learning data-driven discretizations for partial differential equations. *Proceedings of the National Academy of Sciences*, 116(31), 15344–15349.

Beckers, T., Seidman, J., Perdikaris, P., and Pappas, G.J. (2022). Gaussian process port-Hamiltonian systems: Bayesian learning with physics prior. In *61st Conference on Decision and Control (CDC)*, 1447–1453. IEEE.

Cieślik, S. (2021). Mathematical modeling of the dynamics of linear electrical systems with parallel calculations. *Energies*, 14(10), 2930.

Gardner, J.R., Pleiss, G., Bindel, D., Weinberger, K.Q., and Wilson, A.G. (2018). Gpytorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration. In *Advances in Neural Information Processing Systems*.

Hou, Z.S. and Wang, Z. (2013). From model-based control to data-driven control: Survey, classification and perspective. *Information Sciences*, 235, 3–35.

Long, Z., Lu, Y., Ma, X., and Dong, B. (2018). PDE-Net: Learning PDEs from data. In *International conference on machine learning*, 3208–3216. PMLR.

Nageshrao, S.P., Lopes, G.A., Jeltsema, D., and Babuška, R. (2015). Port-Hamiltonian systems in adaptive and learning control: A survey. *IEEE Transactions on Automatic Control*, 61(5), 1223–1238.

Rasmussen, C.E. and Williams, C.K. (2006). *Gaussian processes for machine learning*. MIT press Cambridge.

Stephany, R. and Earls, C. (2022). PDE-READ: Human-readable partial differential equation discovery using deep learning. *Neural Networks*, 154, 360–382.

Van Der Schaft, A. and Jeltsema, D. (2014). Port-Hamiltonian systems theory: An introductory overview. *Foundations and Trends in Systems and Control*, 1(2-3), 173–378.

Wu, K. and Xiu, D. (2020). Data-driven deep learning of partial differential equations in modal space. *Journal of Computational Physics*, 408, 109307.

Zheng, T. and Lin, H. (2022). PDE-based dynamic control and estimation of soft robotic arms. In *61st Conference on Decision and Control (CDC)*, 2702–2707. IEEE.

Zobeiry, N. and Humfeld, K.D. (2021). A physics-informed machine learning approach for solving heat transfer equation in advanced manufacturing and engineering applications. *Engineering Applications of Artificial Intelligence*, 101, 104232.