physNODE: Fusion of data and expert knowledge for modeling dynamical systems Supplements

Leon Lettermann, Alejandro Jurado, Timo Betz, Florentin Wörgötter, and Sebastian Herzog*
Third Institute of Physics - Biophysics,
Georg-August Universität Göttingen,
Göttingen, Deutschland 37077

(Dated: May 31, 2023)

I. IMPLEMENTATION DETAILS OF PHYSNODE

PhysNODE aims to make the adjoint method easily accessible for researchers who want to find parameters describing some ODE system. Therefore, the goal is to provide a software tool that requires only minimal information to be entered, essentially the EOM of the system under consideration. Consequently, the Jacobians in the augmented system Eq. 11,12 have to be derived internally and, additionally, this step needs to be very performant, as it will be called many times while solving the augmented system. Further aspects include possibilities to run simulations in order to validate the fitting accuracy and fitting multiple experiments with a single model.

An important ingredient for achieving the aforementioned goals is the use of JAX [1] as a computational framework. JAX is a python library developed for machine learning applications as an open-source project with support from Google Research. It is built for just-in-time compiling its functions using an Accelerated Linear Algebra (XLA) compiler, a compiler for efficient computation with backends for GPUs and TPUs developed to speed up TensorFlow code, but JAX is built more consequently than TensorFlow to suit and utilize XLA's capabilities. In particular, JAX employs a pure function formalism and is described as a set of composable transformations of functions, including

- `jit`: just-in-time compilation,
- `grad`: autodifferentiation,
- `vmap`: vectorization.

Combination of `grad` (or the lower level `vjp`) and `jit` applied to the EOM yield the performant Jacobians we need, while the `vmap` transformation is used to support multi-experiment fitting. The restriction that the provided EOM has to be compatible with JAX is mitigated by the JAX.numpy module, which rebuilds the popular and well-documented NumPy API. A further advantage is the concept of PyTrees, nested combinations of lists, tuples, dictionaries, and arrays, which are used as type for parameters and system states, providing freedom to organise these. Finally, JAX automatically detects possibly present GPUs or even TPUs, and the code will be compiled and executed using the best available option.

Apart from the data, the input has to be provided as a `define_system` function, the details of which are given in the physNODE Cookbook (see Fig. III D). Passed a number of keywords (`kwargs_sys`) defining system properties, it returns four functions:

- `gen_y0()`: Generating an initial state.
- `gen_params()`: Generating a set of parameters.
- `eom`: The systems Equation of motion, above f.
- `loss`: The loss function, above L.

The state and parameters returned by the first two functions can be any PyTree, usually a dictionary is clearest.

The physNODE module includes the `EquationNODE` class, wrapping the generation of the augmented system Eq. 11,12, the `PhysNodeDataset` class to combine this equations with data and the `simple_simulation` function, which automatically simulated data and sets up a `PhysNodeDataset` object.

The `train_physnode` function takes the gradients delivered by the augmented system to perform training using the well-established ADAM optimizer [2]. The hyperparameters of this optimizer are a learning rate, a learning rate decay controlling exponential decay of the learning rate as well as the ADAM internal constants b_1 and b_2, all of which can be specified via `kwargs_NODE`. Other optimizers can be submitted via respective `kwargs_NODE`. For each of `params`, `iparams` (cf. Section II A) and possibly $\mathtt{y}_0$ that should be fitted a separate ADAM optimizer is used, the hyperparameters of which can independently be adjusted. The augmented method is executed neglecting this dependence, only at the end of each epoch are the gradient contributions generated by the explicit derivative of the loss with respect to the parameters added.

* sherzog3@gwdg.de

In Section C. the idea to solve the system itself backward alongside adjoint state and computed gradient to avoid the necessity to save many intermediate states was introduced. In practice, time asymmetric systems may thwart this idea. One such bad, but unfortunately common, example are diffusion systems. While quite helpful in stabilizing the forward pass, solving a diffusive system backward is equivalent to anti-diffusion, enlarging initial inaccuracies exponentially and thereby rendering the backward pass highly unstable. To handle this, physNODEs is implemented in such a way that the system state is stored at each observation time $\hat{t}_i$, allowing backpropagation between observation times without becoming too unstable. Further stabilization can be achieved by storing additional back up states in between two observation times, the number of which is specified by the keyword 'N_backups'. In order not to save all backup states simultaneously, they are created by an additional forward pass only once their respective time interval is concerned.

Most often the loss function will work such that the current solution is compared to a given solution, e.g. with a mean squared error loss, and the direction of change for optimization enters the augmented system Eq. 11,12 via the Jacobian $\mathcal{J}_{L,y}$.

II. REMARKS AND SPECIFICS

Two potential problems currently exist:

1. There is currently no easy way to install JAX on non-Linux systems, and even there it does not have a built-in CUDA installation, as e.g. PyTorch or Tensorflow, meaning CUDA has to be installed manually. For this reason, we provide a Apptainer recipe to set up a container with CUDA and JAX installed, see `https://gitlab.gwdg.de/sherzog3/physnode.git`. Alternatively, one have to install JAX and its dependencies manually: The installation instruction for JAX and dependencies can be found at `https://github.com/google/jax#installation`
2. Also, the standard ODE solver used is a mixed fourth/fifth-order Runge-Kutta algorithm with Dormand Prince stepsize adaption. Other solvers can be used, but have to be compatible with JAX.

A. Working with physNODE

In the following we would like to provide a few tips that might be helpful when implementing own cases with physNODE:

The shapes of the PyTrees returned by `gen_params()` and `gen_y0()` have to match whatever `loss` and `eom` expect, meaningful values are only required if simulations should be generated. The `gen_params()` function actually has to return three PyTrees, for three different types of parameters called `params`, `iparams` and `exparams`. The first two are the unknown parameters that should be fitted and differ only in multi-experiment situations, where `params` are universal parameters and `iparams` are fitted individually for each experiment. The external parameters `exparams` are assumed to be known, but with individual values specified for each submitted experiment, hence they cannot be built into the system's definition (which is universal for all experiments). For more complicated systems, `define_system` may have subroutines describing parts or steps of the EOM. Those can be returned in an otherwise empty dictionary as a fifth return value, such that they can be accessed later.

Usually `EquationNODE` should not be called directly, but is automatically included in `PhysNodeDataset`. To setup a `PhysNodeDataset`, the `define_system` function and dedicated `kwargs_sys`, as well as data to be fitted and `t_evals`, an array containing the times at which the data was observed, have to be passed. Lastly, a second keyword dictionary, `kwargs_NODE`, contains the settings for the NODE and optimization process. Mandatory are the learning rate ('lr') and number of epochs ('epochs'), but a number of properties and behaviours can be controlled through other keywords. The `simple_simulation` function requires the same information, apart from data, which is generated by randomly taken initial conditions and parameters according to `gen_y0` and `gen_params`. Once a dataset has been generated either with data or a simulation, training can be performed by calling the `train_physnode` function with the dataset as argument.

Often one might want to further restrict or influence the training process. One possibility is to specify boundaries, e.g. to avoid the system reaching unstable parameter values. Another popular choice is weight decay. To include it, in contrast to the above derivation we allow parameter-dependent term in the loss function $L\left(y_{(\hat{t}_0)}, ..., y_{(\hat{t}_N)}, p\right)$.

A typical weight decay would now be included by adding a L_2-norm of the parameters to the loss function, but it allows for arbitrary complex terms guiding the training to follow specific requirements.

For long time series or very inaccurate initial parameters, eventually, the current solution will have diverged so far from the target trajectory, that this comparison is meaningless. This can especially be expected to happen faster for chaotic systems. A very easy solution is only to use a fraction of the data, cutting the time evolution to a sufficiently stable regime.[3] A second possibility is to specify an exponential decay of gradients and the adjoint state applied during backward solving, such that

contributions at later times have lesser influence.[4] The most powerful approach is to divide the time series into many short segments and reload a new initial condition at the beginning of each segment disregarding at what state the previous segment ended up. To conveniently use this, the 't_reset_idcs' keyword in `kwargs_NODE` can be passed a tuple of integers, which are understood as indices of observation times in `t_evals`. At each indicated time the system is reset to the estimated state, taken from the target trajectory. The initial state is enlarged by an additional axis running overall initial conditions at different times, such that if training of initial conditions is active all different initial conditions are trained simultaneously. The `simple_simulation` function is helpful for easy testing, playing around, and having fun, but serves an important purpose in using physNODE to gain information of data. As with any tool from the realm of machine and deep learning, interpretation of the results needs care and vigilance. In this setting, the question to be answered is if found parameters, which yield a good fit of observed and reproduced system behaviour, are necessarily the true parameters. The main purpose of the simulation feature is to generate several systems with plausible trajectories. After the same fitting procedure as for actual data is applied, the found parameters can be compared to those of the simulation to validate the method. The other way round simulations can also be used before fitting data to establish a protocol and hyperparameters. If parameters do not agree even though the final loss was very small, probably some parameters are redundant allowing for different parameters describing the same dynamics, or the loss function doesn't capture all relevant parts of the system.

III. SUPPLEMENTARY FIGURES AND VIDEOS

A. Performance benchmark

The only model presented, in this work, that offers a natural way to increase the size of the system is the repulsive N-body interaction (Sec. I B Hence this system is used for comparison.

Results: The presented framework performs well for all system sizes tested, converging faster or comparable to alternatives (Fig. 1). Although physNODE was designed with large systems and many parameters in mind, and Ma *et al.* [6] found that for smaller systems direct autodifferentiation can be faster, physNODE provides excellent performance and can be applied to many systems out of the box.

B. Figure SF1: Initial conditions for AP Model

In Figure 2 the different initial field for BOCF and AP model are detailed.

BOCF	D	k_s	k_{so}	τ_{fi}	τ_{si}	τ_{so1}	θ_v	u_s	u_{so}	u_u
Rec.	0.2	2.102	2.017	0.111	2.96	43.2	0.299	0.909	0.66	1.583
Truth	0.2	2.099	2.000	0.110	2.87	43.0	0.300	0.909	0.65	1.580

TABLE I. The recovered and true values for the parameters for the BOCF model. Units are as in the original publications (BOCF: [5]).

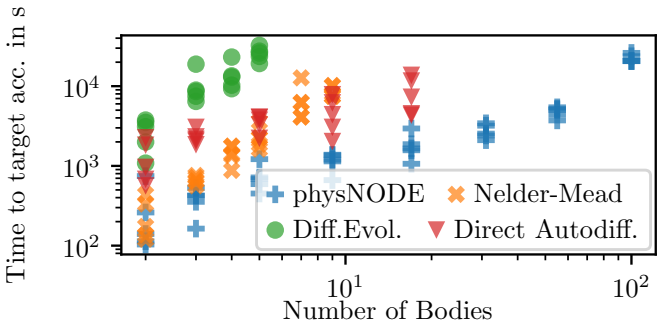

FIG. 1. Performance comparison based on Sec. I B.

C. Video V1: BOCF and AP Models

Results of BOCF model. a-d: Excitation field u, value given by colour bar on the right. a: Recovered BOCF field after 1.5 s. b: Difference between recovery in a and target. c: Truth after 0.5 s. d: Recovery using the Aliev-Panfilov model. e: Mean absolute error of recovery using BOCF and AP models. f: Error in peak occurrence time averaged over all pixels.

D. Video V2: Rayleigh Bénard Convection

Results for 2D Rayleigh-Bénard Convection. a-c: Temperature perturbation. a: Truth, b: Recovery, c: Difference. d: Mean absolute error between recovered and true temperature perturbation. e: Pixelvalue of center pixel.

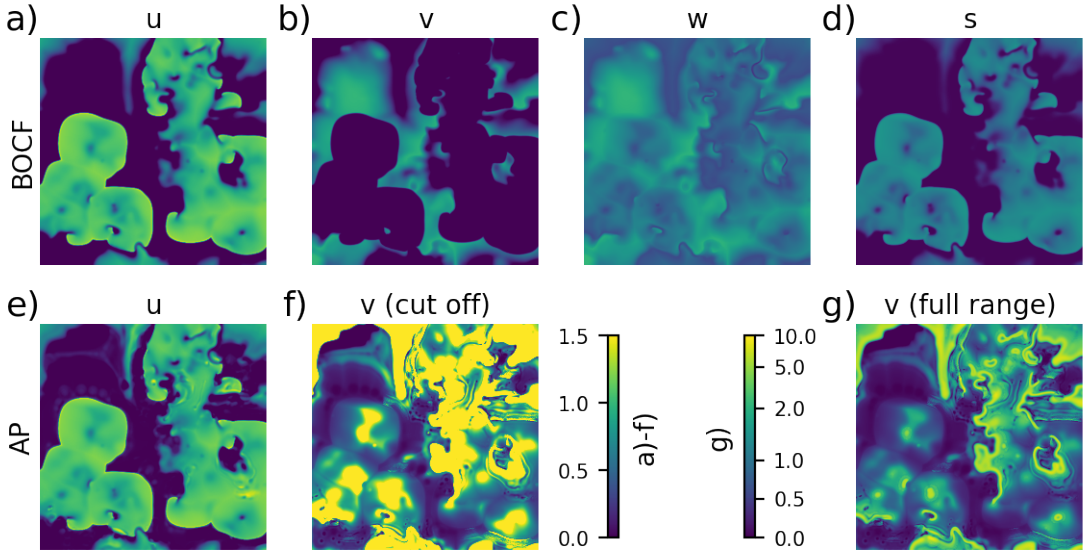

FIG. 2. a)-d): The fixed initial conditions used for generating the BOCF trajectories, u (a) being the excitation field and v,w, and s (b-d) the different gating variables. e)-g): The resulting optimized initial conditions for the AP model in order to reproduce the BOCF model. Excitation field u (e) and the single gating field v, displayed once (f) with the common color scheme used for a)-f) and once with a separate one to show the full range (g).

[1] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, JAX: composable transformations of Python+NumPy programs (2018).

[2] D. P. Kingma and J. Ba, Adam: A method for stochastic optimization, in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, edited by Y. Bengio and Y. LeCun (2015).

[3] `kwargs_NODE` keyword: 't_stop_idx'.

[4] `kwargs_NODE` keyword: 'time_decay'.

[5] A. Bueno-Orovio, E. M. Cherry, and F. H. Fenton, Minimal model for human ventricular action potentials in tissue, Journal of theoretical biology **253**, 544 (2008).

[6] Y. Ma, V. Dixit, M. J. Innes, X. Guo, and C. Rackauckas, A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions, in *2021 IEEE High Performance Extreme Computing Conference (HPEC)* (IEEE, 2021) pp. 1–9.

The physNODE Cookbook

0. Install physNODE and JAX

Make sure JAX and if you want to use GPUs a supported CUDA driver is installed, as well as physNODE and its dependencies. An installation guide is provided in the git-repository, `https://gitlab.gwdg.de/sherzog3/physnode.git`.

1. Define your System

Our example system is $\frac{d}{dt}pop = a \cdot pop + b$, where pop is some scalar population and a and b are the parameters we want to find. We assume the initial population, a and b to be bounded below by zero and above by some maximum specified in `kwargs_sys`.

```
import numpy as np
import jax.numpy as jnp
from jax import jit
def define_system(**kwargs_sys):
    p_max = kwargs_sys['p_max']
    a_max = kwargs_sys['a_max']
    b_max = kwargs_sys['b_max']

    def gen_y0():
        ini_pop = np.random.rand()*p_max
        return {'population':ini_pop}

    def gen_params():
        a = np.random.rand()*a_max
        b = np.random.rand()*b_max
        return {'a':a, 'b':b}, {}, {}

    @jit
    def eom(y, t, params, iparams, exparams):
        pop = y['population']
        a, b = params['a'], params['b']
        return {'population':a*pop+b}

    @jit
    def loss(ys, params, iparams,
                    exparams, targets):
        pop = ys['population']
        t_pop = targets['population']
        return jnp.mean((pop-t_pop)**2)

    return eom, loss, gen_params, gen_y0, {}
```

The second and third dictionary of `gen_params` are `iparams` and `exparams` we do not have in this simple example. The first two functions can be arbitrary, the `eom` and `loss` functions have to be implemented using the jax libraries.

2. Set up a simulation

To set up a simulation we define the dictionaries `kwargs_sys` and `kwargs_NODE` as well as the times `t_evals` at which we assume to observe our system. The keyword 'N_sys' gives the number of copies in terms of multi-experiment fitting, here we consider only one system.

```
from physNODE.Framework import
    simple_simulation, train_physNode
kwargs_sys = {'p_max': 2,'a_max': 1,
              'b_max': 3, 'N_sys': 1}
kwargs_NODE = {'lr':1e-2, 'epochs':50}
t_evals = np.linspace(0,50,10)
dataset = simple_simulation(define_system,
                            kwargs_sys,
                            t_evals,
                            kwargs_NODE)
```

3. Train a simulation

The easy following command trains our simulation and prints the true params in comparison to the found ones:

```
_ = train_physNode(dataset)
print('True params: ', dataset.params)
print('Found params: ', dataset.params_train)
```

4. Including Data

To include data, we bring it in the same form as the shape of the state given by `gen_y0()`, but with two additional leading axes. The first counts the different experiments, and has length one here, the second runs over time points and has the same length as `t_evals`.

```
from physNODE.Framework import PhysNodeDataset
data # Observation of population, shape (10,)
targets = {'population':data.reshape((1,10))}
dataset2 = PhysNodeDataset(define_system,
                           kwargs_sys,
                           t_evals,
                           targets,
                           kwargs_NODE)
```

This new dataset can be trained just the same, although we can not print the true parameters in comparison as we do not know them.