Pacific Northwest
NATIONAL LABORATORY

PNNL-34895

Domain Aware Deep-learning Algorithms Integrated with Scientific-computing Technologies (DADAIST)

September 2023

Jan Drgona
Aaron Tuor
James Koch
Madelyn Shapiro
Ethan King
Draguna Vrabie

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
email: orders@ntis.gov <https://www.ntis.gov/about>
Online ordering: http://www.ntis.gov

PNNL-34895

Domain Aware Deep-learning Algorithms Integrated with Scientific-computing Technologies (DADAIST)

September 2023

Jan Drgona
Aaron Tuor
James Koch
Madelyn Shapiro
Ethan King
Draguna Vrabie

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Abstract

This technical report summarized the contribution of the DADAIST project funded by the Data Model Convergence Initiative via the Laboratory Directed Research and Development (LDRD) investments at Pacific Northwest National Laboratory (PNNL). Specifically, we report the development of the NeuroMANCER (Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations), a new open-source Scientific Machine Learning library for formulating and solving parametric constrained optimization problems, physics-informed system identification, and parametric optimal control problems. NeuroMANCER is using differentiable programming to combine modern data-driven models and optimization modeling language into a coherent algorithmic and software framework. NeuroMANCER is a Pytorch-based framework and adopts much of its philosophy focused on research and development, rapid prototyping, and streamlined deployment. Strong emphasis is given to extensibility, interoperability with the PyTorch ecosystem, and quick adaptability to custom domain problems. Neuromancer repository contains a comprehensive library of differentiable modules, including custom activation functions, matrix factorizations, deep learning architectures, neural differential equations, differential equation solvers, implicit layers such as iterative solvers, high-level API for symbolic expressions, API for modeling and control of dynamical systems, and extensive set of tutorial code examples in the form of python scripts and jupyter notebooks.

Acknowledgments

This research was partially supported by the Data Model Convergence (DMC) and Mathematics for Artificial Reasoning in Science (MARS) initiatives via the Laboratory Directed Research and Development (LDRD) investments at Pacific Northwest National Laboratory (PNNL), by the U.S. Department of Energy, through the Office of Advanced Scientific Computing Research's “Data-Driven Decision Control for Complex Systems (DnC2S)” project, and through the Energy Efficiency and Renewable Energy, Building Technologies Office under the “Dynamic decarbonization through autonomous physics-centric deep learning and optimization of building operations” and the “Advancing Market-Ready Building Energy Management by Cost-Effective Differentiable Predictive Control” projects. PNNL is a multi-program national laboratory operated for the U.S. Department of Energy (DOE) by Battelle Memorial Institute under Contract No. DE-AC05-76RL0-1830.

Contents

Figures

1.0 Neuromancer Requirements

This section describes the requirements for the NeuroMANCER library.

1.1 Functional Requirements

Generally, functional requirements are expressed in the form "system must do <requirement>".

User Interface requirements
NeuroMANCER as a Scientific Machine Learning framework should provide a user-friendly interface for formulating and instantiating differentiable programming problems for:

- parametric constrained optimization
- physics-informed dynamical systems modeling
- parametric constrained optimal control

The NeuroMANCER framework should construct differentiable programs based on following input-output specifications:

- Input: high-level syntax and algebraic symbolic language for defining the objectives, constraints, and trainable components of the differentiable programming problem.
- Output:Instantiated differentiable programming problem represented by Pytorch nn.Module class.

Solvers requirements
NeuroMANCER should come with a set of solvers and meta-heuristics for hyperparameter optimization providing an automated solution for complex differentiable programming problems.

Hard constraints guarantees requirements
NeuroMANCER should provide a set of model architectures and solution methods that can guarantee the satisfaction of hard constraints with user-defined precision.

HPC support requirements
NeuroMANCER should provide a set of templates for dispatch, training, and analysis of the differentiable programming problems using HPC machines such as GPU clusters.

1.2 Non-Functional Requirements

Non-functional requirements take the form "system shall be <requirement>."

User experience requirements
NeuroMANCER should be user-friendly and intuitive with emphasis on users from various engineering domains such as mechanical, electrical, chemical, and control engineering domains. NeuroMANCER should be a tool for easy prototyping and execution of developed programs. NeuroMANCER should be well documented with README, user manual, code tutorials, and docstrings compiled by pydot.

Architecture requirements
NeuroMANCER's object-oriented framework should be modular and provide a set of modeling abstractions and templates for constructing the problems mentioned above. Each class should come with standardized Type defined input-output specifications.

Interaction requirements

- Interoperability with the PyTorch ecosystem. Since NeuroMANCER is built on top of Pytorch; its APIs shall be designed in a way to allow easy integration of third-party model architectures implemented as Pytorch nn.Modules into NeuroMANCER's computational graphs.
- Interoperability with constrained optimization frameworks such as CVXPY, Pyomo, or CasADi needs to be developed. This will include: 1) using constrained optimization solvers as safety filters in the online deployment of models trained in NeuroMANCER, 2) extraction of computational graphs or constrained optimization problem formulations into CVXPY, Pyomo, or CasADi.
- NeuroMANCER should also come with a base class abstraction that will allow for easy implementation of new solvers.

Development and Maintanance requirements

NeuroMANCER shall be easily extensible with new model architectures, solvers, and features. NeuroMANCER development shall be safe with protected branches, reviewed merge requests, and automated pytest executions.

Open-source requirements

NeuroMANCER shall be a free open-source repository.

Reliability

NeuroMANCER shall be reliable and robust. NeuroMANCER installation needs to be tested on all supported operating systems. All open-source examples and tutorials need to be bug-free and tuned, and verified on all operating systems. Open-sourced model architectures and solvers need to provide robust convergence across datasets and hyperparameter scenarios.

1.3 User Diagram

The intended use of the library is illustrated in the use case diagram is shown in Figure 1.

List of Neuromancer actors

- Developer
- End-user
- Computing platform (CPU, GPU, HPC cluster)

List of intended use cases

- Execute and modify tutorial and example scripts.
- Create new tutorials and example scripts.
- Formulate constrained optimization problems in high level symbolic language
- Construct differentiable programs of parametric optimization problems in a form of symbolic computational graphs
- Visualize computational graphs of differentiable programs
- Construct AggregateLoss class.
- Construct Trainer class.
- Solve differentiable parametric programming programs (constrained machine learning, constrained optimization, system identification, and control) with sampling-based automatic differentiation

- Evaluate and Visualize the performance of the obtained parametric solution on a given test case (e.g., prediction accuracy of system identification task, or closed-loop control performance of trained control policy)
- Construct DataLoader with dataset.
- Impement new Dataset Class.
- Impement new Callback Class.
- Modify Trainer class.
- Dispatch distribution of experiments with specified hyperparameters (e.g., training models on GPU cluster with hyperparameter search)
- Implement new component architectures (e.g., custom neural ordinary differential equation architectures)
- Implement new solvers (e.g., Augmented Lagrangian method, or Interior point algorithm)
- Expand or modify core Neuromancer library (e.g., modify forward pass method of the Problem class)
- Implement new visualization capability.

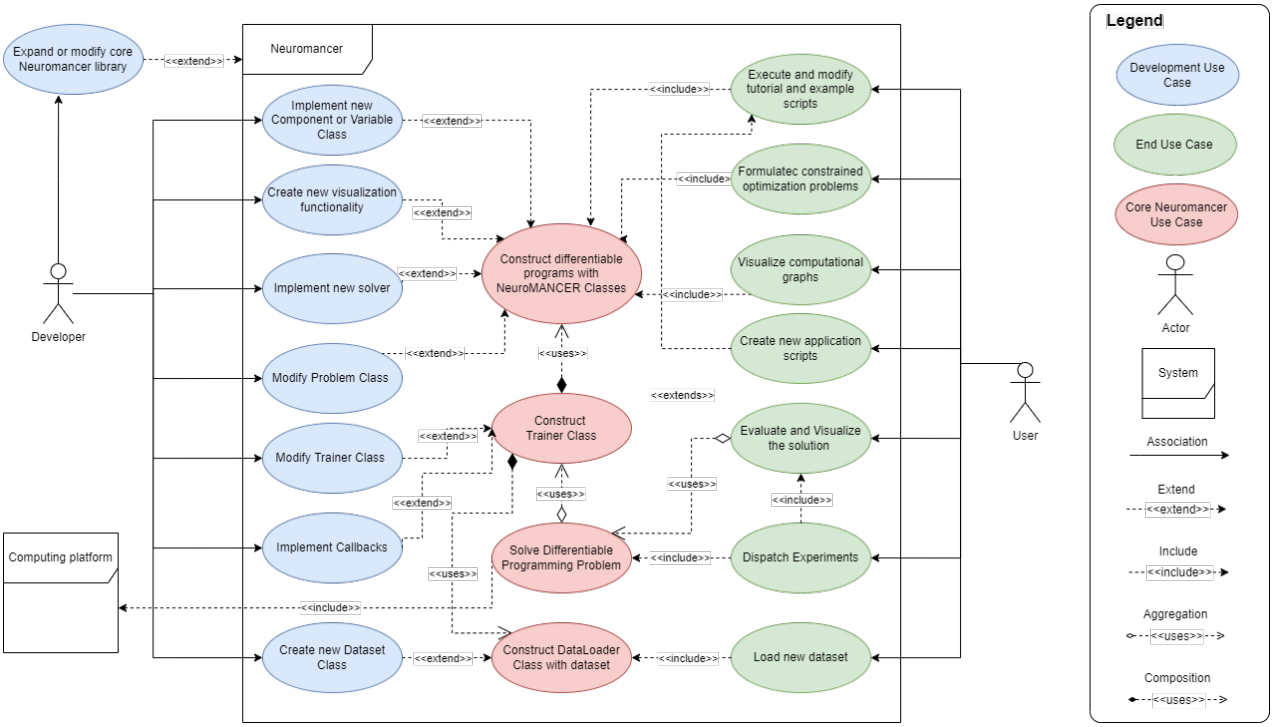

Figure 1. Neuromancer use case diagram.

2.0 Neuromancer Architecture and API

Architecture

We compactly represent Neuromancer's architecture using UML Class diagram shown in Figure 2.

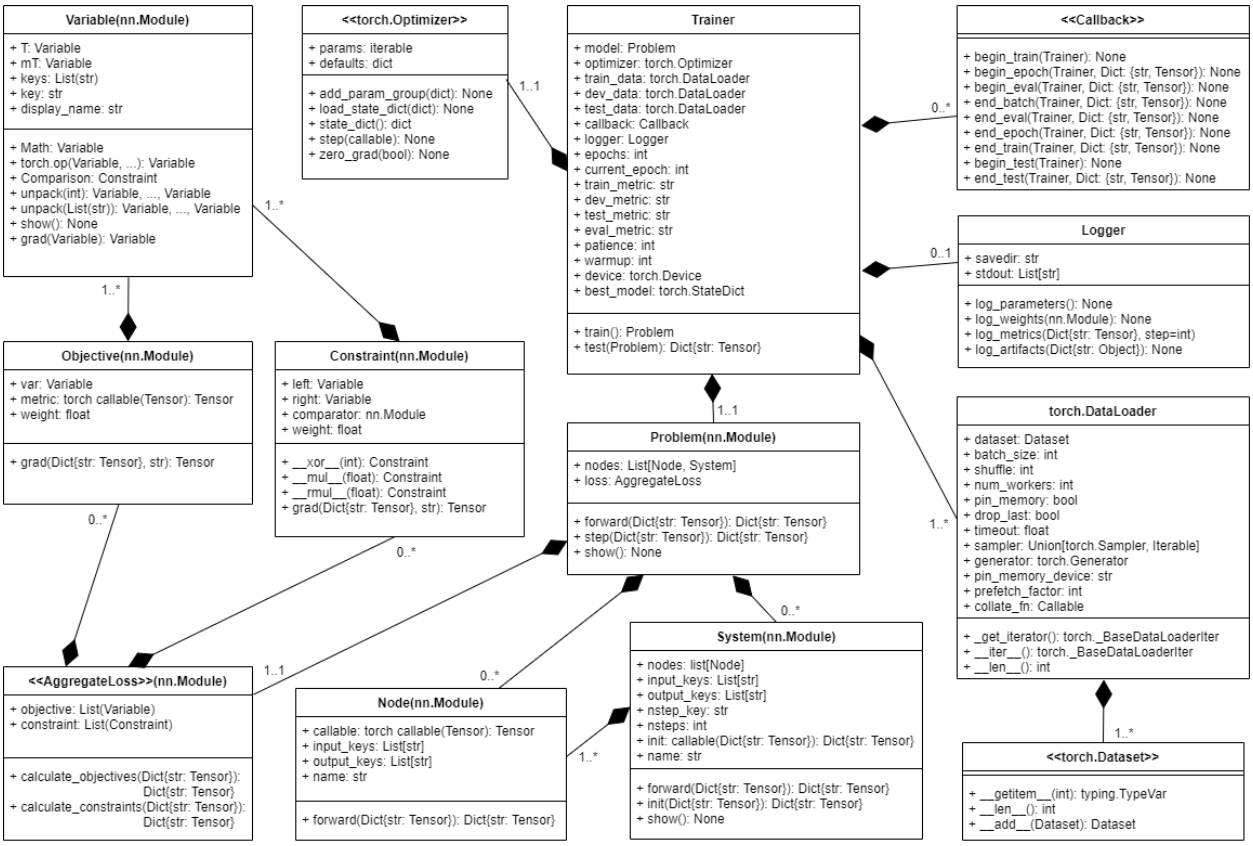

Figure 2. Neuromancer UML diagram.

API specifications of Neuromancer classes and functions can be found online at: https://pnnl.github.io/neuromancer/

Neuromancer contains the following classes:

- **Trainer**: Class encapsulating boilerplate PyTorch training code. Training procedure is somewhat extensible through methods in Callback objects associated with training and evaluation waypoints.Trainer is instantiated with given Problem class and Pytorch Dataloader classes storing Neuromancer Datasets.

- **Callback**: Class for versatile behavior in the Trainer object at specified checkpoints. Allows the user to customize training, evaluation, and testing phases of the optimization algorithm.

- **Logger**: class for saving arguments, metrics, and artifacts (images, video) into specified directory. Also allows to control the verbosity of print statements during training.

- **Dataset**: class compatible with neuromancer Trainer based on parent Pytorch Dataset class. Implements static, sequence, and graph structured datasets.

- **DataLoader**: class from Pytorch combines a dataset and a sampler, and provides an iterable over the given dataset.

- **Problem**: class is similar in spirit to a nn.Sequential module. However, by concatenating input and output dictionaries for each component class we can represent arbitrary directed acyclic computation graphs. Problem class represents complete differentiable constrained optimization problem with scalar valued training metric suitable for gradient-based optimization via backpropagation algorithm.

- **AggregateLoss:** abstract class for calculating constraints, objectives, and aggregate loss values suitable for automatic differentiation via backpropagation algorithm. Implements different loss aggregation methods such as: Penalty Method, Barrier Method, or Augmented Lagrangian methodm.

- **Variable** class is an abstraction that allows for the definition of constraints and objectives with some nice syntactic sugar. When a Variable object is called given a dictionary a pytorch tensor is returned, and when a Variable object is subjected to a comparison operator a Constraint is returned. Mathematical operators return Variables which will instantiate and perform the sequence of mathematical operations.

- **Node** abstract class allows to wrap arbitrary nn.Modules in Pytorch into symbolic representation of the computational graph. Component is mapping input keys onto output keys representing symbolic variables of the computational graph whose forward pass is defined by Pytorch nn.Module.

- **Constraint** is a class constructed by a composition of Variable objects using comparative infix operators, '<', '>', '==', '<=', '>=' and '*' to weight loss component and '\wedge' to determine l-norm of constraint violation in determining loss. A Constraint has the intuitive syntax for defining constraints of optimization problems via Variable objects.

- **Objective** is a class constructed via neuromancer Variable object and given metric with forward pass that evaluates metric as torch function on Variable values. Objective allows to create Loss function terms directly from instantiated Variables.

3.0 Neuromancer Methods and Algorithms

This section documents differentiable programming methods and algorithms for solution of: 1) parametric constrained optimization problems, 2) physics-constrained system identification problems, and 3) parametric optimal control problems.

3.1 Learning to Solve Constrained Optimization Problems

https://github.com/pnnl/neuromancer/tree/master/examples/parametric_programming

Learning Solutions to Constrained Optimization Problems is a set of methods that use machine learning to learn the solutions (explicit solvers) to optimization problems. Constrained optimization problems where the solution x depends on the varying problem parameters ξ are called parametric programming problems. Neuromancer allows you to formulate and solve a broad class of parametric optimization problems via the Differentiable Programming (DP) paradigm. Hence, we call the approach Differentiable Parametric Programming (DPP). Specifically, Neuromancer allows you to use automatic differentiation (AD) in PyTorch to compute the sensitivities of such constrained optimization problems w.r.t. their parameters. This allows you to leverage gradient-based optimizers (e.g., stochastic gradient descent) to obtain approximate solutions to constrained parametric programming problems via for semi-supervised offline learning. The main advantage of this offline DPP-based solution compared to classical optimization solvers (e.g., IPOPT) is faster online evaluation, often obtaining orders of magnitude speedups.

Imitation Learning vs Differentiable Parametric Programming

Recent years have seen a rich literature of deep learning (DL) models for solving the constrained optimization problems on real-world tasks such as power grid, traffic, or wireless system optimization. Earlier attempts simply adopt imitation learning (i.e., supervised learning) to train function approximators via a minimization of the prediction error using labeled data of pre-computed solutions using iterative solvers (i.e. IPOPT). Unfortunately, these models can hardly perform well on unseen data as the outputs are not trained to satisfy physical constraints, leading to infeasible solutions. To address the feasibility issues, existing methods have been imposing constraints on the output space of deep learning models for a subsequent differentiation using AD tools. These differentiable programming-based methods, also called end-to-end learning with constraints or learning to optimize, directly consider the original objectives and constraints in the DL training process without the need of expert labeled data. The following figure conceptually demonstrated the difference between supervised imitation learning and unsupervised Differentiable Parametric Programming (DPP) which solution is obtained by differentiating the objectives and constraints of the parametric optimization problem.

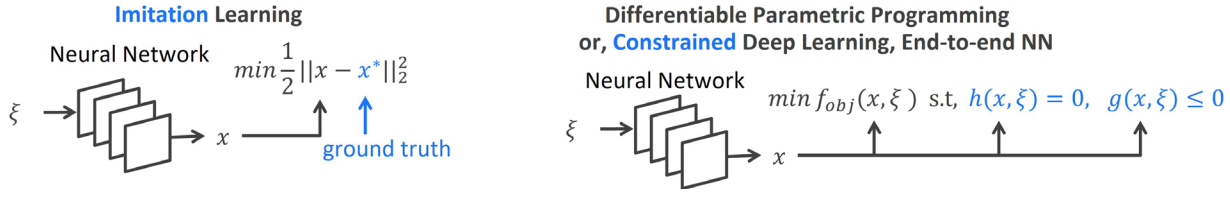

Figure 3. Imitation learning VS end-to-end learning using Differentiable Parametric Programming.

DPP problem formulation

A generic formulation of the DPP is given in the form of a parametric optimization problem:

$$\min_{\Theta} \mathcal{L}_{\text{obj}} = \min_{\Theta} \frac{1}{m} \sum_{i=1}^{m} \mathbf{f}(\mathbf{x}^i, \boldsymbol{\xi}^i) \tag{1a}$$

$$\text{s.t. } \mathbf{g}(\mathbf{x}^i, \boldsymbol{\xi}^i) \leq 0, \ \mathbf{h}(\mathbf{x}^i, \boldsymbol{\xi}^i) = 0, \tag{1b}$$

$$\mathbf{x}^i = \boldsymbol{\pi}_{\Theta}(\boldsymbol{\xi}^i), \ \boldsymbol{\xi}^i \in \Xi, \ \forall i \in \mathbb{N}_1^m \tag{1c}$$

Where Ξ represents the sampled dataset, and $\boldsymbol{\xi}^i$ represents i-th batch of the sampled problem data. The vector $\mathbf{x}^i$ represents optimized variables that minimize the loss function while satisfying a set of inequality and equality constraints ((1b)). The map $\pi_{\Theta}(\boldsymbol{\xi}^i)$ is given by a deep neural network parametrized by Θ and represents the parametric solution of the DPP problem.

There are several ways in which we can enforce the constraints satisfaction while learning the solution π_Θ(ξ) of the differentiable constrained optimization problem (1). The simplest approach is to penalize the constraints violations by augmenting the loss function L (1a) with the penalty loss function given as:

$$\mathcal{L}_{\text{con}} = \frac{1}{m} \sum_{i=1}^{m} \left(Q_g ||\texttt{ReLU}(\mathbf{g}(\mathbf{x}^i, \boldsymbol{\xi}^i))||_l + Q_h ||\mathbf{h}(\mathbf{x}^i, \boldsymbol{\xi}^i)||_l \right) \tag{2}$$

Where l denotes the norm type and Q_g, Q_h being the corresponding weight factors. The overall loss then becomes $\mathcal{L}_{\text{penalty}} = \mathcal{L}_{\text{obj}} + \mathcal{L}_{\text{con}}$.

DPP problem solution

The main advantage of having a differentiable objective function and constraints in the DPP problem formulation (1) is that it allows us to use automatic differentiation to directly compute the gradients of the parametric solution map π_Θ(ξ). In particular, by representing the problem (1) as a computational graph and leveraging the chain rule, we can directly compute the gradients of the loss function L w.r.t. the solution map weights Θ as follows:

$$\nabla_{\Theta} \mathcal{L}_{\text{penalty}} = \frac{\partial \mathcal{L}_{\text{obj}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \Theta} + \frac{\partial \mathcal{L}_{\text{con}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \Theta} = \frac{\partial \mathcal{L}_{\text{obj}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \Theta} + \frac{\partial \mathcal{L}_{\text{con}}(\mathbf{x}, \boldsymbol{\xi})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \Theta} \tag{3}$$

Where $\frac{\partial \mathbf{x}}{\partial \Theta}$ represent partial derivatives of the neural network solution map w.r.t. its weights that are typically being computed in deep learning applications via backpropagation. The advantage of having gradients (3) is that it allows us to use scalable stochastic gradient optimization algorithms such as AdamW [1] to solve the corresponding DPP problem (1) by direct offline optimization of the neural network. In practice, we can compute the gradient of the DPP problem by using automatic differentiation frameworks such as Pytorch [2].

The gradient-based solution of the DPP problem is summarized in the following Algorithm:

Algorithm 1 Differentiable Parametric Programming Algorithm.

input training dataset Ξ of sampled problem parameters.
input differentiable solution map architecture $\pi_\Theta(\boldsymbol{\xi})$ parametrized by Θ.
input differentiable constrained optimization objective $\mathbf{f}(\mathbf{x}, \boldsymbol{\xi})$ and constraints $\mathbf{g}(\mathbf{x}, \boldsymbol{\xi}) \leq 0,\ \mathbf{h}(\mathbf{x}, \boldsymbol{\xi}) = 0$
input DPP loss function aggregator $\mathcal{L}_{\text{penalty}}$
input optimizer $\mathbb{O}$
differentiate DPP loss $\mathcal{L}_{\text{penalty}}$ to obtain the parameter gradients $\nabla_\Theta \mathcal{L}_{\text{penalty}}$ of the solution map $\pi_\Theta(\boldsymbol{\xi})$
learn solution map $\pi_\Theta(\boldsymbol{\xi}^i)$ parametrized by Θ via optimizer $\mathbb{O}$ using gradient $\nabla_\Theta \mathcal{L}_{\text{penalty}}$
return trained parametric solution $\pi_\Theta(\boldsymbol{\xi})$

Related literature

- A. Agrawal, et al., Differentiable Convex Optimization Layers, 2019
- F. Fioretto, et al., Predicting AC Optimal Power Flows: Combining Deep Learning and Lagrangian Dual Methods, 2019
- S. Gould, et al., Deep Declarative Networks: A New Hope, 2020
- P. Donti, et al., DC3: A learning method for optimization with hard constraints, 2021
- J. Kotary, et al., End-to-End Constrained Optimization Learning: A Survey, 2021
- M. Li, et al., Learning to Solve Optimization Problems with Hard Linear Constraints, 2022
- R. Sambharya, et al., End-to-End Learning to Warm-Start for Real-Time Quadratic Optimization, 2022

3.2 Learning Physics-informed Neural Models of Dynamical Systems

https://github.com/pnnl/neuromancer/tree/master/examples/system_identification

Differentiable models such as Neural ordinary differential equations (NODEs) or neural state space models (NSSMs) represent a class of black box models that can incorporate prior physical knowledge into their architectures and loss functions. Examples include structural assumption on the computational graph inspired by domain application, or structure of the weight matrices of NSSM models, or networked NODE architecture illustrated in Figure 4. Differentiability of NODEs and NSSMs allows us to leverage gradient-based optimization algorithms for learning the unknown parameters of these structured digital twin models from observational data of the real system.

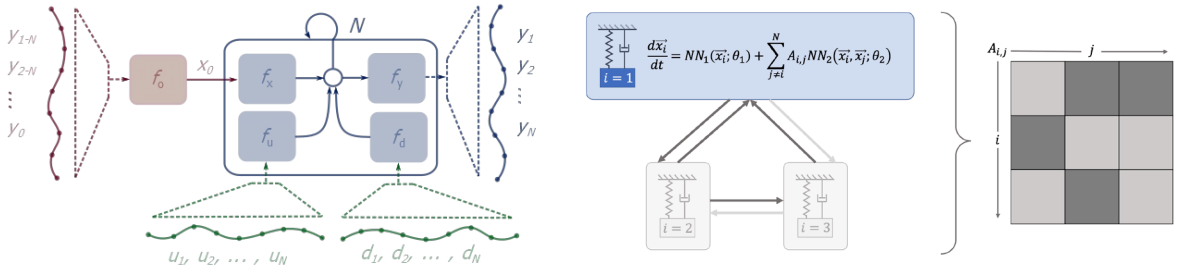

Figure 4. Structural priors in neural models of dynamical systems.

System Identification Problem

Consider the non-autonomous partially observable nonlinear dynamical system:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)), \tag{4a}$$
$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t)), \tag{4b}$$
$$\mathbf{x}(t_0) = \mathbf{x}_0, \quad \mathbf{u}(t_0) = \mathbf{u}_0 \tag{4c}$$

where $\mathbf{y}(t) \in \mathbb{R}^{n_y}$ are measured outputs, and $\mathbf{u}(t) \in \mathbb{R}^{n_u}$ represents controllable inputs or measured disturbances affecting the system dynamics.

We assume access to a limited set of system measurements in the form of tuples, each of which corresponds to the input-output pairs along sampled trajectories with temporal gap Δ. That is, we form a dataset:

$$S = \{(\mathbf{u}_t^{(i)}, \mathbf{y}_t^{(i)}), (\mathbf{u}_{t+\Delta}^{(i)}, \mathbf{y}_{t+\Delta}^{(i)}), \ldots, (\mathbf{u}_{t+N\Delta}^{(i)}, \mathbf{y}_{t+N\Delta}^{(i)})\}, \tag{5}$$

where i = 1, 2, . . . , n represents up to n different batches of input-output trajectories with N -step time horizon length. The primary objective of the physics-constrained system identification is to construct structured digital twin models and learn their unknown parameters from the provided observation data to provide accurate and robust long-term prediction capabilities.

Our recent development work in Neuromancer has given us the capability to learn dynamical systems of the form:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t)$$

where x(t) is the time-varying state of the considered system, u(t) are system control inputs, and f is the state transition dynamics. This modeling strategy can be thought of as an equivalent method to Neural Ordinary Differential Equations1, whereby an ODE of the above forms is fit to data with a universal function approximator (e.g. deep neural network) acting as the state transition dynamics. To train an appropriate RHS, Chen et al. utilize a continuous form of the adjoint equation; itself solved with an ODESolver. Instead, we choose to utilize the autodifferentiation properties of PyTorch to build differentiable canonical ODE integrators.

We wish to test the capability of this methodology in a variety of situations and configurations. Of particular interest is the predictive capability of this class of methods compared with Neural State Space Models and other traditional “black-box” modeling techniques.

Before moving on, it is important to note that there are two dominant neural ODE packages freely available. The first is DiffEqFlux.jl developed and maintained by SciML within the Julia ecosystem. The second is torchdyn which lives within the PyTorch ecosystem. Both packages are well-documented and have become established in application-based research literature.

System Identification Solution

The primary learning objective is to minimize the mean squared error, Ly, between predicted values and the ground truth measurements for the N -step prediction horizon:

$$\mathcal{L}_y = \frac{1}{n \cdot N \cdot n_y} \sum_{i=1}^{n} \sum_{t=1}^{N} \sum_{j=1}^{n_y} (\hat{\mathbf{y}}_{t,j}^{(i)} - \mathbf{y}_{t,j}^{(i)})^2 \tag{11}$$

where n_y is the dimension of the observations $\mathbf{y}_t$ and predictions $\hat{\mathbf{y}}$, and n is the number of N-step long training trajectories.

The system identification objective (11) can be augmented with various kind of physics-informed soft constraints. In the following we enumerate a few examples. First, we apply inequality constraints on output predictions during training in order to promote the boundedness and convergence of our dynamical models:

$$\mathbf{s}^{\underline{\mathbf{y}}} = \max(0, -\hat{\mathbf{y}} + \underline{\mathbf{y}}) \tag{12a}$$

$$\mathbf{s}^{\overline{\mathbf{y}}} = \max(0, \hat{\mathbf{y}} - \overline{\mathbf{y}}) \tag{12b}$$

$$\mathcal{L}_y^{\text{con}} = \frac{1}{n_y} \sum_{i=1}^{n_y} (\mathbf{s}_i^{\underline{\mathbf{y}}} + \mathbf{s}_i^{\overline{\mathbf{y}}}) \tag{12c}$$

To promote continuous trajectories of our dynamics models, we optionally apply a state smoothing loss which minimizes the mean squared error between successive predicted states:

$$\mathcal{L}_{dx} = \frac{1}{(N-1)n_x} \sum_{t=1}^{N-1} \sum_{i=1}^{n_x} (\mathbf{x}_t^{(i)} - \mathbf{x}_{t+1}^{(i)})^2 \tag{16}$$

We include constraints penalties as additional terms to the optimization objective 14, and further define coefficients, Q∗ as hyperparameters to scale each term in the multi-objective loss function

$$\mathcal{L} = Q_y \mathcal{L}_y + Q_{dx} \mathcal{L}_{dx} + Q_y^{\text{con}} \mathcal{L}_y^{\text{con}} \tag{17}$$

The physics-constrained system identification training with differentiable digital twin models is summarized in the following Algorithm:

Algorithm 2 Physics-constrained system ID algorithm with differentiable models.

1: **input** training dataset S (5) of sampled input output trajectories.
2: **input** differentiable digital twin model architecture M, e.g. (9), (11), (12), parametrized by θ.
3: **input** System ID loss $\mathcal{L}$ (17)
4: **input** optimizer $\mathbb{O}$
5: **differentiate** System ID loss $\mathcal{L}$ to obtain the parameter gradients $\nabla_\theta \mathcal{L}$ of the system models M
6: **learn** system dynamics model M parametrized by θ via optimizer $\mathbb{O}$ using gradient $\nabla_\theta \mathcal{L}$
7: **return** trained system dynamics model M

Related literature

- James Koch, Zhao Chen, Aaron Tuor, Jan Drgona, Draguna Vrabie, Structural Inference of Networked Dynamical Systems with Universal Differential Equations, arXiv:2207.04962, (2022)

- Drgoňa, J., Tuor, A. R., Chandan, V., & Vrabie, D. L., Physics-constrained deep learning of multi-zone building thermal dynamics. Energy and Buildings, 243, 110992, (2021)
- E. Skomski, S. Vasisht, C. Wight, A. Tuor, J. Drgoňa and D. Vrabie, "Constrained Block Nonlinear Neural Dynamical Models," 2021 American Control Conference (ACC), 2021, pp. 3993-4000, doi: 10.23919/ACC50511.2021.9482930.
- Skomski, E., Drgoňa, J., & Tuor, A. (2021, May). Automating Discovery of Physics-Informed Neural State Space Models via Learning and Evolution. In Learning for Dynamics and Control (pp. 980-991). PMLR.
- Tuor, A., Drgona, J., & Vrabie, D. (2020). Constrained neural ordinary differential equations with stability guarantees. arXiv preprint arXiv:2004.10883.

3.3 Differentiable Predictive Control

https://github.com/pnnl/neuromancer/tree/master/examples/control

Differentiable predictive control (DPC) method represents a flagship capability of the Neuromancer library. DPC allows us to learn control policy parameters directly by backpropagating model predictive control (MPC) objective function and constraints through the differentiable model of a dynamical system. Instances of a differentiable model include ordinary differential equations (ODEs), including neural ODEs, universal differential equations (UDEs), or neural state space models (SSMs).

The conceptual methodology shown in the figures below consists of two main steps. In the first step, we perform system identification by learning the unknown parameters of differentiable digital twins. In the second step, we close the loop by combining the digital twin models with control policy, parametrized by neural networks, obtaining a differentiable closed-loop dynamics model. This closed-loop model now allows us to use automatic differentiation (AD) to solve the parametric optimal control problem by computing the sensitivities of objective functions and constraints to changing problem parameters such as initial conditions, boundary conditions, and parametric control tasks such as time-varying reference tracking.

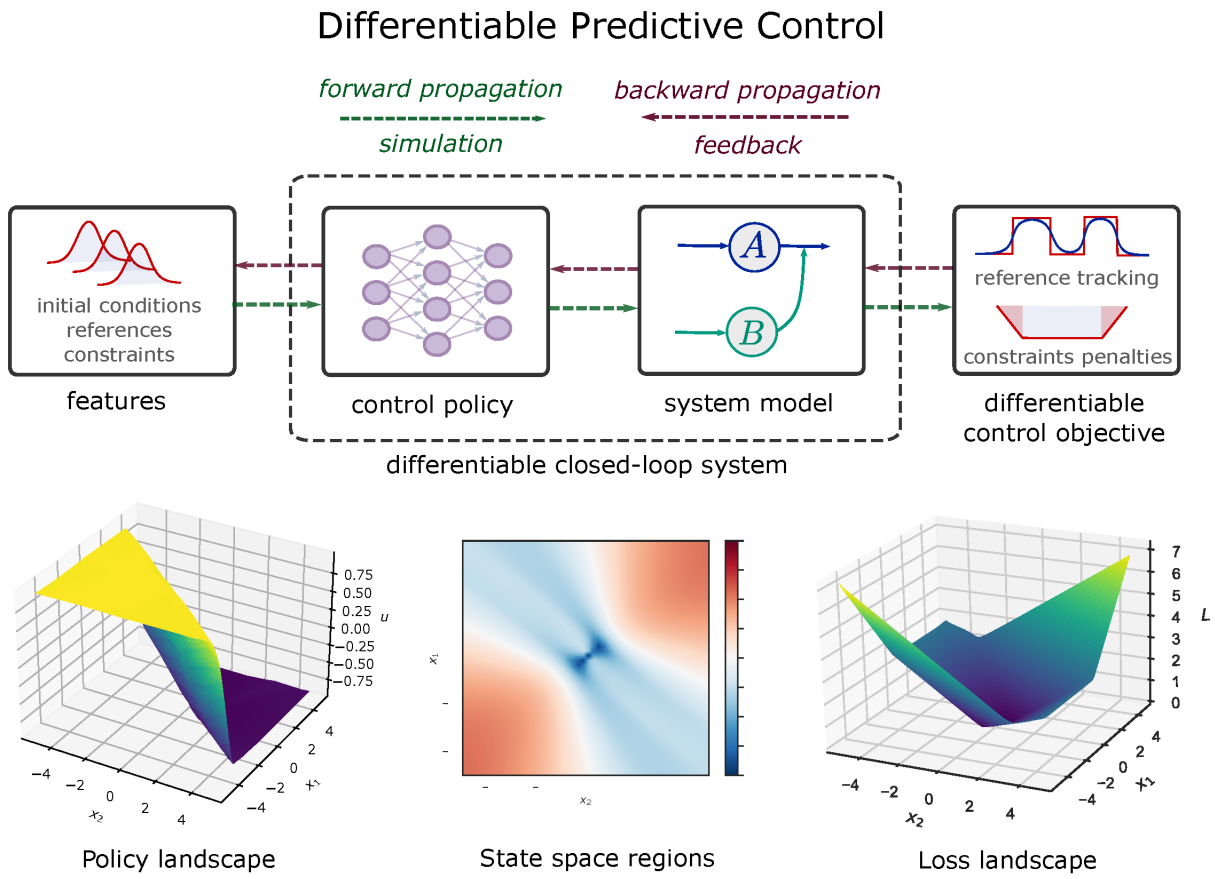

Figure 5. Conceptual DPC methodology. Simulation of the differentiable closed-loop system dynamics in the forward pass is followed by backward pass computing direct policy gradients for policy optimization.

DPC problem formulation

Formally we can formulate the DPC problem as a following parametric optimal control problem:

$$\min_{\mathbf{W}} \frac{1}{mN} \sum_{i=1}^{m} \sum_{k=0}^{N-1} \left(\ell_{\texttt{MPC}}(\mathbf{x}_k^i, \mathbf{u}_k^i, \mathbf{r}_k^i) + p_x(h(\mathbf{x}_k^i, \mathbf{p_h}_k^i)) + p_u(g(\mathbf{u}_k^i, \mathbf{p_g}_k^i)) \right) \tag{15a}$$

$$\text{s.t. } \mathbf{x}_{k+1}^i = \mathbf{f}(\mathbf{x}_k^i, \mathbf{u}_k^i), \;\; k \in \mathbb{N}_0^{N-1} \tag{15b}$$

$$\mathbf{u}_k^i = \pi_{\mathbf{W}}(\mathbf{x}_k^i, \boldsymbol{\xi}_k^i) \tag{15c}$$

$$\mathbf{x}_0^i \in \mathbb{X} \subset \mathbb{R}^{n_x} \tag{15d}$$

$$\boldsymbol{\xi}_k^i = \{\mathbf{r}_k^i, \mathbf{p_h}_k^i, \mathbf{p_g}_k^i\} \in \Xi \subset \mathbb{R}^{n_\xi} \tag{15e}$$

The DPC loss function is composed of the parametric MPC objective $\ell_{\texttt{MPC}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{r}_k) : \mathbb{R}^{n_x+n_u+n_r} \to \mathbb{R}$, and penalties of parametric constraints $p_x(h(\mathbf{x}_k, \mathbf{p_h}_k) : \mathbb{R}^{n_h+n_{p_h}} \to \mathbb{R}$, and $p_u(g(\mathbf{u}_k, \mathbf{p_g}_k)) : \mathbb{R}^{n_g+n_{p_g}} \to \mathbb{R}$. The MPC objective $\ell_{\texttt{MPC}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{r}_k)$ is a differentiable function representing the control performance metric such as the following reference tracking with control action minimization in the quadratic form:

$$\ell_{\texttt{MPC}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{r}_k) = ||\mathbf{x}_k - \mathbf{r}_k||_{Q_r}^2 + ||\mathbf{u}_k||_{Q_u}^2 \tag{16}$$

with reference states $\mathbf{r}_k$, and $\|\mathbf{a}\|_Q^2 = \mathbf{a}^T Q \mathbf{a}$ the weighted squared 2-norm. The control parameters $\boldsymbol{\xi}$ are sampled from the synthetically generated training dataset Ξ, where m represents the total number of parametric scenario samples, and i denotes the index of the sample.

DPC problem solution

The main advantage of having a differentiable closed-loop dynamics model, control objective function, and constraints in the DPC problem formulation is that it allows us to use automatic differentiation (backpropagation through time) to directly compute the policy gradient. In particular, by representing the problem (15) as a computational graph and leveraging the chain rule, we can directly compute the gradients of the loss function w.r.t. the policy parameters W as follows:

$$\begin{aligned} \nabla_{\mathbf{W}} \mathcal{L}_{\texttt{DPC}} = \frac{\partial \ell_{\texttt{MPC}}(\mathbf{x}, \mathbf{u}, \mathbf{r})}{\partial \mathbf{W}} + \frac{\partial p_x(h(\mathbf{x}, \mathbf{p_h}))}{\partial \mathbf{W}} + \frac{\partial p_u(g(\mathbf{u}, \mathbf{p_g}))}{\partial \mathbf{W}} = \\ \frac{\partial \ell_{\texttt{MPC}}(\mathbf{x}, \mathbf{u}, \mathbf{r})}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} + \frac{\partial \ell_{\texttt{MPC}}(\mathbf{x}, \mathbf{u}, \mathbf{r})}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} + \\ \frac{\partial p_x(h(\mathbf{x}, \mathbf{p_h}))}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} + \frac{\partial p_u(g(\mathbf{u}, \mathbf{p_g}))}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{W}} \end{aligned} \tag{17}$$

Where $\frac{\partial \mathbf{u}}{\partial \mathbf{W}}$ represent partial derivatives of the neural policy outputs w.r.t. its weights that are typically being computed in deep learning applications via backpropagation. The advantage of having gradients (17) is that it allows us to use scalable stochastic gradient optimization algorithms such as AdamW [1] to solve the corresponding parametric optimal control problem (15) by direct offline optimization of the neural control policy. In practice, we can compute the gradient of the DPC problem by using automatic differentiation frameworks such as Pytorch [2].

The DPC policy optimization algorithm is summarized in the following figure. The differentiable system dynamics model is required to instantiate the computational graph of the DPC problem The policy gradients ∇L are obtained by differentiating the DPC loss function L over the distribution of initial state conditions and problem parameters sampled from the given training datasets X and Ξ, respectively. The computed policy gradients now allow us to perform direct policy optimization via a gradient-based optimizer O. Thus the presented procedure introduces a generic approach for data-driven solution of model-based parametric optimal control problem (15) with constrained neural control policies.

Algorithm 3 DPC policy optimization.

1: **input** training datasets of sampled initial conditions $\mathbb{X}$ and problem parameters Ξ
2: **input** differentiable digital twin model
3: **input** DPC loss $\mathcal{L}_{\text{DPC}}$
4: **input** optimizer $\mathbb{O}$
5: **differentiate** DPC loss $\mathcal{L}_{\text{DPC}}$ to obtain the policy gradient $\nabla_{\mathbf{W}}\mathcal{L}_{\text{DPC}}$
6: **learn** policy $\pi_{\mathbf{W}}$ via optimizer $\mathbb{O}$ using gradient $\nabla_{\mathbf{W}}\mathcal{L}_{\text{DPC}}$
7: **return** optimized policy $\pi_{\mathbf{W}}$

From a reinforcement learning (RL) perspective, the DPC loss L can be seen as a reward function, with ∇L representing a deterministic policy gradient. The main difference compared with actor-critic RL algorithms is that in DPC the reward function is fully parametrized by a closed-loop system dynamics model, control objective, and constraints penalties. The model-based approach avoids approximation errors in reward functions making DPC more sample efficient than model-free RL algorithms.

DPC problem architecture

The forward pass of the DPC computational graph is conceptually equivalent with a single shooting formulation of the model predictive control (MPC) problem. The resulting structural equivalence of the constraints of classical implicit MPC in a dense form with DPC is illustrated in the following figure. Similarly, to MPC, in the open-loop rollouts, the explicit DPC policy generates future control action trajectories over N-step prediction horizon given the feedback from the system dynamics model. Then for the closed-loop deployment, we adopt the receding horizon control (RHC) strategy by applying only the first-time step of the computed control action.

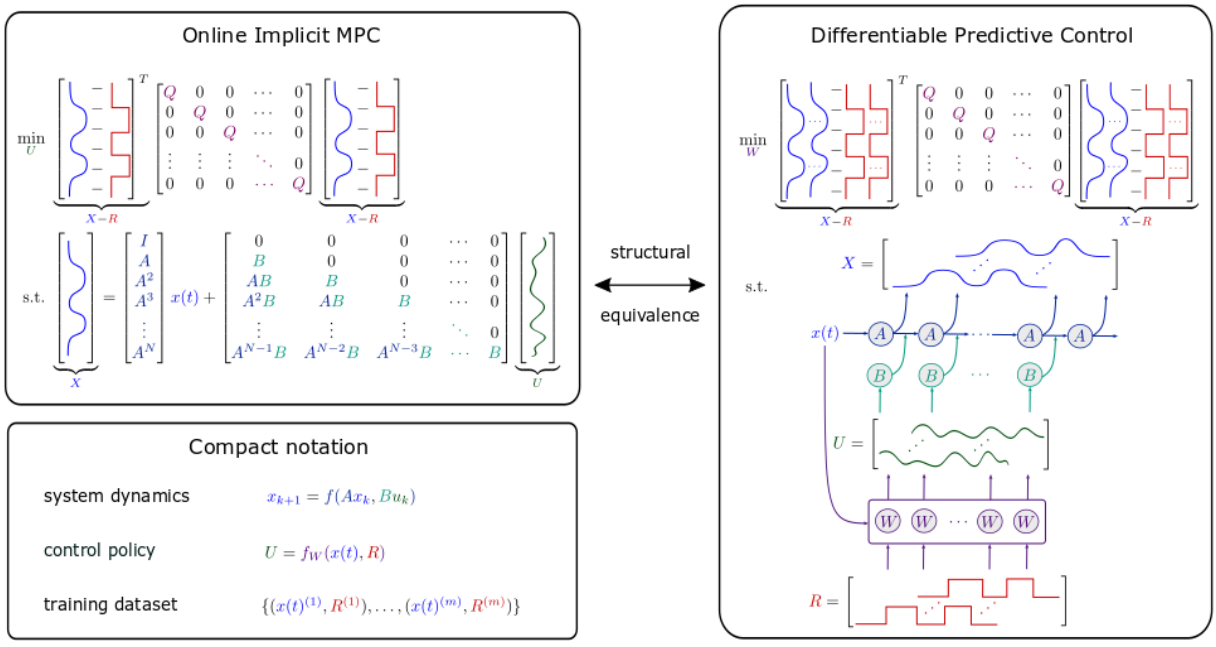

Figure 6. Structural equivalence of DPC architecture with MPC constraints.

Related literature

[1] Drgona, J., Tuor, A., & Vrabie, D., Learning Constrained Adaptive Differentiable Predictive Control Policies With Guarantees, arXiv preprint arXiv:2004.11184, 2020

[2] Drgona, Jan, et al. "Differentiable Predictive Control: An MPC Alternative for Unknown Nonlinear Systems using Constrained Deep Learning." Journal of Process Control Volume 116, August 2022, Pages 80-92

[3] Drgoňa, J., Tuor, A., Skomski, E., Vasisht, S., & Vrabie, D. Deep Learning Explicit Differentiable Predictive Control Laws for Buildings. IFAC-PapersOnLine, 54(6), 14-19., 2021

[4] Ján Drgoňa, Sayak Mukherjee, Aaron Tuor, Mahantesh Halappanavar, Draguna Vrabie, Learning Stochastic Parametric Differentiable Predictive Control Policies, IFAC-PapersOnLine, Volume 55, Issue 25, 2022, Pages 121-126, ISSN 2405-8963

[5] Sayak Mukherjee, Ján Drgoňa, Aaron Tuor, Mahantesh Halappanavar, Draguna Vrabie, Neural Lyapunov Differentiable Predictive Control, IEEE Conference on Decision and Control Conference 2022

[6] Wenceslao Shaw Cortez, Jan Drgona, Aaron Tuor, Mahantesh Halappanavar, Draguna Vrabie, Differentiable Predictive Control with Safety Guarantees: A Control Barrier Function Approach, IEEE Conference on Decision and Control Conference 2022

[7] Ethan King, Jan Drgona, Aaron Tuor, Shrirang Abhyankar, Craig Bakker, Arnab Bhattacharya, Draguna Vrabie, Koopman-based Differentiable Predictive Control for the Dynamics-Aware Economic Dispatch Problem, American Control Conference (ACC) 2022

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354

1-888-375-PNNL (7665)

www.pnnl.gov