Proceedings of Machine Learning Research vol 242:541–553, 2024 6th Annual Conference on Learning for Dynamics and Control

Learning for CasADi
Data-driven Models in Numerical Optimization

Tim Salzmann TIM.SALZMANN@TUM.DE
Jon Arrizabalaga JON.ARRIZABALAGA@TUM.DE
Technical University of Munich, Germany
Joel Andersson JOEL@JAEANDERSSON.COM
Freelance software developer and consultant, Madison (WI), United States
Marco Pavone PAVONE@STANFORD.EDU
Stanford University, Stanford (CA), United States
and **Markus Ryll** MARKUS.RYLL@TUM.DE
Munich Institute of Robotics and Machine Intelligence, Technical University of Munich, Germany

Editors: A. Abate, M. Cannon, K. Margellos, A. Papachristodoulou

Abstract

While real-world problems are often challenging to analyze analytically, deep learning excels in modeling complex processes from data. Existing optimization frameworks like CasADi facilitate seamless usage of solvers but face challenges when integrating learned process models into numerical optimizations. To address this gap, we present the *Learning for CasADi* (L4CasADi) framework, enabling the seamless integration of PyTorch-learned models with CasADi for efficient and potentially hardware-accelerated numerical optimization. The applicability of *L4CasADi* is demonstrated with two tutorial examples: First, we optimize a fish's trajectory in a turbulent river for energy efficiency where the turbulent flow is represented by a PyTorch model. Second, we demonstrate how an implicit Neural Radiance Field environment representation can be easily leveraged for optimal control with *L4CasADi*.

L4CasADi, along with examples and documentation, is available under MIT license at

https://github.com/Tim-Salzmann/l4casadi

Keywords: optimization; machine learning; control systems; data-driven control

1. Introduction

Accurate mathematical problem formulation is at the core of every numerical optimization procedure. While many real-world problems are hard to formulate analytically, data-driven methods, especially deep learning methods, thrive in modeling complex processes from data. Efficiently integrating learned process models into numerical optimizations is challenging, as data-driven models and numerical optimization come with their respective tools and characteristics: Deep learning models, commonly constructed in PyTorch Paszke et al. (2019), TensorFlow Abadi et al. (2016), or JAX Bradbury et al. (2018), leverage first-order optimization methods (backpropagation) in time-expensive offline training on a large amount of data. Solutions to complex problems can be learned from data without defining any prior structure on the task.

Once trained, however, their inference is fast and can be further accelerated on dedicated hardware (GPUs). Numerical optimization techniques, in contrast, employ second-order methods, primarily interior point and sequential quadratic programming, to formulate optimization problems, potentially subject to constraints. Solving these problems necessitates formulating a specific problem structure and proficiently selecting a suitable solver algorithm (IPOPT, SNOPT, qpOASES, OSQP, ECOS, etc.[1]). Aiming to remove the overhead of adapting the syntax for each solver, multiple frameworks have appeared allowing seamless usage across them. Within convex problems, CVXPY Diamond and Boyd (2016), is a widely used Python wrapper. On the other hand, Pyomo Bynum et al. (2021), AMPL Fourer et al. (1989), or CasADi Andersson et al. (2019) allow for a wider range – including non-convex – of optimization problems and solvers, including commercial ones as MOSEK MOSEK ApS (2023) or Gurobi Gurobi Optimization LLC (2023).

Thus, both PyTorch and CasADi have established themselves as prominent tools within their respective research domains, owing to their comprehensive functionalities, user-friendliness, and adaptability. However, the increasing importance of data-driven approaches in optimization poses a challenge for CasADi as it lacks native support for learned functions.

We seek to close this gap by presenting the *Learning for CasADi* (L4CasADi) framework, which enables the seamless integration of learned PyTorch models with the numerical optimization framework CasADi Andersson et al. (2019). *L4CasADi* enables hardware acceleration for learned components in a CasADi optimization formulation and can generate such problems as plain C/C++ code for efficient solution. By bringing this functionality to CasADi, tools building upon CasADi (such as acados Verschueren et al. (2021)), as well as the large number of products already using CasADi, and the numerical optimization community at large can benefit from the same data-driven modeling capabilities.

2. L4CasADi - Syntax and Usage

L4CasADi was designed with three key desiderata in mind: (I) Simplicity for the user, (II) generalizability across PyTorch model architectures, and (III) efficiency in runtime. We will provide an insight into how these desiderata manifest within the user experience of *L4CasADi*.

Similarly to PyTorch, *L4CasADi* models are initially constructed in Python. Defining an *L4CasADi* model in Python given a pre-defined PyTorch model is as easy as

```
import l4casadi as l4c
# Construct L4CasADi Model from PyTorch Model
l4casadi_model = l4c.L4CasADi(
   pyTorch_model,
   device='cpu',      # Device in ['cpu', 'gpu', 'mps']
   name='l4casadi_f'  # Unique name
)
```

where the architecture of the PyTorch model is unrestricted and large models can be accelerated with dedicated hardware. Once an *L4CasADi* model is defined it can be employed in a variety of settings, as will be outlined in the following.

1. Wächter and Biegler (2006); Gill et al. (2005); Ferreau et al. (2014); Stellato et al. (2020); Domahidi et al. (2013)

2.1. Python

Once defined, an *L4CasADi* model can be seamlessly integrated with a CasADi symbolic graph within CasADi's Python interface. On the first call, *L4CasADi* will automatically generate C++ code and compile the *L4CasADi* model for runtime efficiency.

```
# Use L4CasADi Model in CasADi Symbolic Graph
y: casadi.MX = l4casadi_model(x: casadi.MX)
```

The resulting symbolic output variable can be included in any further CasADi operations. This seamless integration empowers the utilization of CasADi's extensive toolkit to formulate and solve optimization problems involving *L4CasADi* models.

```
# Minimize the L4CasADi Model using IPOPT
nlp = {'x': x, 'f': y}
solver = casadi.nlpsol("solver", "ipopt", nlp)
sol = solver()
```

2.2. Standalone Application

To use an *L4CasADi* model outside of the defining Python routine, the *L4CasADi* model can also be explicitly built into an *L4CasADi Shared Library Function*.

```
# Build the L4CasADi Model as L4CasADi Shared Library Function
l4casadi_model.build(x: casadi.MX) # Creates libl4casadi_f.so
```

The shared library can easily be integrated into other programming environments, e.g. Matlab and C/C++.

Matlab A built *L4CasADi* model can be loaded and used in Matlab using CasADi's Matlab interface.

```
% Load L4CasADi Model from dynamic library
l4casadi_model = casadi.external('l4casadi_f', 'libl4casadi_f.so');
% Use loaded L4CasADi Model in CasADi symbolic Graph
y: casadi.MX = l4casadi_model(x: casadi.MX)
```

C/C++ A built (*L4CasADi Shared Library Function*) can be dynamically loaded from any C/C++ program. Additionally, *L4CasADi* provides the *L4CasADi* model as auto-generated C++ code which can be included in any C/C++ source project.

```
# Generate the L4CasADi Model as C++ Source Files.
l4casadi_model.generate(x: casadi.MX) # Generates l4casadi_f.cpp
```

2.3. Naïve L4CasADi

While *L4CasADi* was designed with efficiency in mind by internally leveraging torch's C++ interface (see Section 4), this can still result in overhead, which can be disproportionate for

small, simple models. Thus, *L4CasADi* additionally provides a *NaiveL4CasADiModule* which directly recreates the PyTorch computational graph using CasADi operations and copies the weights — leading to a pure C computational graph without context switches to torch. However, this approach is limited to a small predefined subset of PyTorch operations — only Multi-Layer-Perceptron style models and inference on CPU are supported.

```
naive_mlp: l4c.NaiveL4CasADiModule = l4c.naive.MultiLayerPerceptron(
   in_features,
   hidden_features,
   out_features,
   hidden_layers,
   activation='Tanh'
)
l4c_model = l4c.L4CasADi(naive_mlp)
y: casadi.MX = l4c_model(x: casadi.MX)
```

3. Tutorial Examples

To showcase the practical application of the proposed framework and how it opens up new research avenues at ease, we present two illustrative case studies. First, we formulate a trajectory generator that finds the minimum energy path for a fish swimming upstream in a turbulent river. The second case study demonstrates how *L4CasADi* facilitates the incorporation of cutting-edge computer vision models into optimization problems. In this case, we optimize a collision-free trajectory through an implicit environment representation given as a Neural Radiance Field (NeRF). These two case studies are distinguished by their simplicity and ease of comprehension, and thereby, serve as excellent templates for users to adapt to more intricate scenarios. To this end, the code associated with these examples is available alongside *L4CasADi*[2].

3.1. Fish Navigation in Turbulent Flow

The example shows how to design an optimization-based trajectory generator to navigate a turbulent fluid flow. The flow, and thus the fish's dynamics, are modeled by a Neural Network in PyTorch, while the optimal trajectory optimization problem is formulated in CasADi. By using *L4CasADi*, both can be combined and optimized jointly. When doing so, we seek to find the minimum energy trajectory that allows a fish to navigate from the starting point to the goal. For this purpose, the fish needs to swim upstream a river where a circular stone causes the flow to be turbulent.

3.1.1. Problem Formulation and Implementation

We model the fish as a planar point mass actuated by velocity commands under the influence of the river's velocity field:

$$\dot{\boldsymbol{p}}(t) = \boldsymbol{v}(t) + \boldsymbol{v}_{fl}(t, \boldsymbol{p}(t)), \tag{1}$$

where $\boldsymbol{p}(t), \boldsymbol{v}(t) \in R^2$ are the position and velocity of the fish at time t and $\boldsymbol{v}_{fl}(t, \boldsymbol{p}(t)) \in R^2$ is the velocity of the river at time t and position $\boldsymbol{p}(t)$. In simpler terms, the fish's movement

2. https://github.com/Tim-Salzmann/l4casadi/examples

results from a combination of its own effort, and the influence of the turbulent flow. Eq. (1) can be written as a standard nonlinear dynamic system f, defined as

$$\dot{\boldsymbol{x}}(t) = f(\boldsymbol{x}(t),\boldsymbol{u}(t),t) = \boldsymbol{u}(t) + \boldsymbol{v}_{fl}(t,\boldsymbol{x}(t)), \tag{2}$$

whose states and inputs are the fish's position $\boldsymbol{x}(t) = \boldsymbol{p}(t)$ and velocity command $\boldsymbol{u}(t) = \boldsymbol{v}(t)$. To add realism to the problem, we constrain the fish's actuation by imposing box constraints on the velocity commands $\boldsymbol{u} \in [\underline{\boldsymbol{u}}, \bar{\boldsymbol{u}}]$, limit the trajectory to lie within the rivers bounds $[\underline{\boldsymbol{p}}, \bar{\boldsymbol{p}}]$ and constrain the trajectory to dodge the stone generating the turbulent flow.

To compute the trajectory that minimizes the fish's effort to reach the goal while swimming upstream, we formulate the following Nonlinear Program (NLP):

$$\min_{\substack{\boldsymbol{x}_0,\cdots,\boldsymbol{x}_N,\\ \boldsymbol{u}_0,\cdots,\boldsymbol{u}_{N-1}}} \sum_{k=0}^{N-2} \left|\left| \frac{\boldsymbol{u}_{k+1} - \boldsymbol{u}_k}{\Delta t} \right|\right|^2 \tag{3a}$$

$$\text{s.t.} \quad \boldsymbol{x}_0 = \boldsymbol{p}_0, \boldsymbol{x}_N = \boldsymbol{p}_f, \tag{3b}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta t \cdot f(\boldsymbol{x}_k, \boldsymbol{u}_k, t_k), \quad k = 0,\cdots,N-1, \tag{3c}$$

$$\underline{\boldsymbol{u}} \leq \boldsymbol{u}_k \leq \bar{\boldsymbol{u}}, \quad k = 0,\cdots,N-1, \tag{3d}$$

$$\underline{\boldsymbol{p}} \leq \boldsymbol{x}_k \leq \bar{\boldsymbol{p}}, \quad k = 0,\cdots,N, \tag{3e}$$

$$||\boldsymbol{x}_k||^2 \geq r_{st}^2, \quad k = 0,\cdots,N. \tag{3f}$$

where $\boldsymbol{p}_0$ and $\boldsymbol{p}_f$ are the initial and goal positions, Δt is the time step and r_{st} is the radius of the stone located at the origin. The cost function (3a) minimizes the differences in subsequent control (velocity) inputs which effectively minimizes the energy introduced into the system, constraint (3b) sets the starting and ending positions, (3c) enforces the system dynamics defined in (2), (3d) bounds the input commands, (3e) guarantees that the fish remains within the river's bounds and (3f) ensures that the fish does not collide with the stone. Notice that, due to the spatial bounds in (3f) and yet-to-be-defined turbulent flow model $\boldsymbol{v}_{fl}(\cdot)$ within the dynamic model $f(\boldsymbol{x},\boldsymbol{u})$ in (3c), the NLP (3) is nonlinear and non-convex.

L4CasADi allows for implementing the NLP in (3) in CasADi, modeling the turbulent flow $\boldsymbol{v}_{fl}(t,\boldsymbol{p}(t))$ by a Neural Network that has previously been trained in PyTorch. Given the nonlinear and non-convex structure of (3), the state-of-the-art interior-point solver IPOPT Wächter and Biegler (2006) is chosen as a solver.

3.1.2. Outcome and Visualization

The trajectory obtained from solving the NLP (3) is illustrated in Fig. 1 through a series of image sequences. The upper segment of figure (A) displays the magenta trajectory of the fish juxtaposed with the vorticity of the turbulent flow. Meanwhile, the middle segment (B) portrays the velocity field of the turbulent flow. In both instances, the stone is represented by a gray circle. The third segment (C) shows the velocity commands necessary to trace this trajectory. For a better insight, we encourage readers to view the accompanying animations for this example[3].

Fig. 1 reveals that the computed trajectory strategically utilizes the velocity field of the turbulent flow to minimize the energy required for reaching the objective. To put it simply,

3. Further visualizations and animations for the fish example can be found at https://github.com/Tim-Salzmann/l4casadi/blob/main/examples/fish_turbulent_flow/README.md.

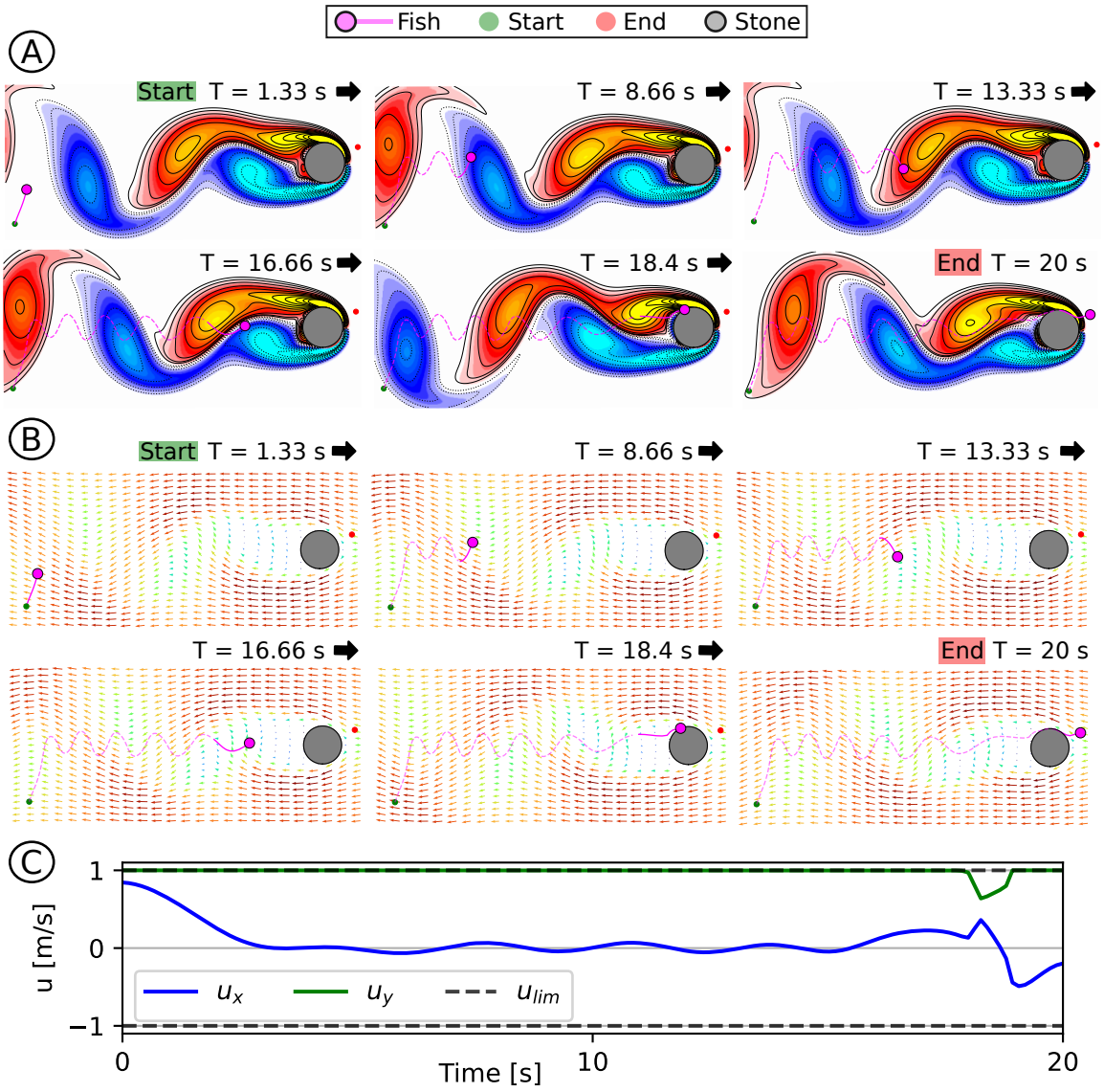


Figure 1: Minimum energy trajectory (magenta) to navigate from the start (green) to the goal (red) in the presence of a turbulent flow represented by its vorticity (A) and velocity field (B). The trajectory generator is implemented in CasADi, while the flow is modeled by a Neural Network in PyTorch[3].

the object behaves like it's riding/surfing the river's current. Examining Fig. 1 enables us to comprehend this phenomenon from three distinct perspectives: Firstly, in (A), it becomes evident that the fish exploits the positive vortices (blue) to ascend and the negative ones (red) to descend. This observation is similarly evident in (B), where the fish adeptly positions itself per the velocity field, enabling it to exert minimal effort in the y-axis direction throughout the majority of the navigation, as depicted by the green line in (C).

3.2. Collision-free Trajectory Optimization in NeRFs

Neural Radiance Fields (NeRFs) are a powerful 3D representation technique that leverages deep learning to reconstruct 3D scenes from a collection of 2D images. NeRFs encode the scene's geometry and appearance into a continuous function, allowing for the rendering of photorealistic images from arbitrary viewpoints. As a byproduct, the densities (translucencies) of the scene's objects at any point in the 3D environment are implicitly captured in the network. In this example, we demonstrate that such state-of-the-art learned models from computer vision research can be easily incorporated into optimization procedures using *L4CasADi*. To showcase this, we present the problem of finding a collision-free trajectory through the densities represented by a learned NeRF, where densities below a predefined threshold are deemed as unobstructed regions within the environment.

3.2.1. Problem Formulation

The trajectory to be planned is assumed to be given by a time-parametric polynomial of degree 9:

$$\boldsymbol{r}(\boldsymbol{c},t) = \sum_{i=0}^{9} \boldsymbol{c}_i t^i \tag{4}$$

where the parametric variable t is time and $\boldsymbol{c} \in \mathbb{R}^{9\times 3}$ are the polynomial's coefficients.

L4CasADi enables the integration of a NeRF as an implicit environment representation into the optimization problem. For this purpose, we define the following function:

$$\rho = f_{\text{NeRF}}(\boldsymbol{p}) \tag{5}$$

where $\rho \in \mathbb{R}^+$ is the density of a location whose Euclidean coordinates are given by $\boldsymbol{p} \in \mathbb{R}^3$, i.e., it returns 0 for obstacle-free space while its value increases as the location becomes occupied.

Having defined the analytical expression of the trajectory in (4) and the NeRF-based environment representation in (5), we formulate a NLP that minimizes the curve's snap, while ensuring that it remains collision-free:

$$\min_{\boldsymbol{c}} \sum_{k=0}^{N} \left|\left| \boldsymbol{r}^{(4)}(\boldsymbol{c},t_k) \right|\right|^2 \tag{6a}$$

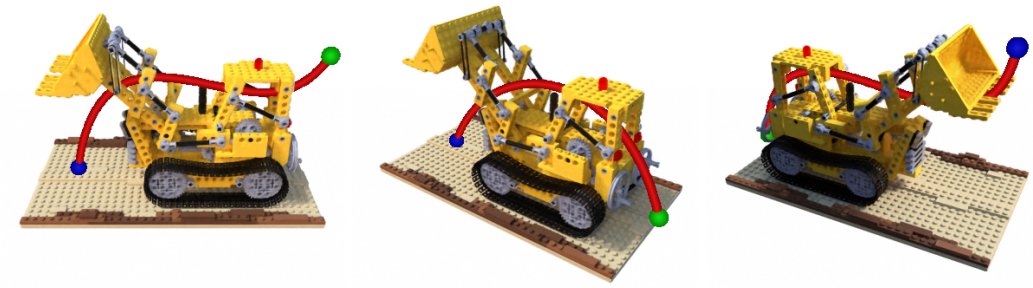
$$\text{s.t.} \quad \boldsymbol{r}(\boldsymbol{c},0) = \boldsymbol{p}_0, \boldsymbol{r}^{(1)}(\boldsymbol{c},0) = \boldsymbol{0}, \boldsymbol{r}^{(2)}(\boldsymbol{c},0) = \boldsymbol{0}, \tag{6b}$$

$$\boldsymbol{r}(\boldsymbol{c},T) = \boldsymbol{p}_f, \boldsymbol{r}^{(1)}(\boldsymbol{c},T) = \boldsymbol{0}, \boldsymbol{r}^{(2)}(\boldsymbol{c},T) = \boldsymbol{0}, \tag{6c}$$

$$\bar{\rho} > f_{\text{NeRF}}(\boldsymbol{r}(\boldsymbol{c},t_k)), \qquad k = 0,\cdots,N, \tag{6d}$$

where $r^{(n)}(\cdot)$ is the n-th time derivative of eq. (4), N is the number of evaluation points, T is the total time assigned to the trajectory. Constraints (6b) and (6c) define the starting and ending conditions and (6d) ensures that the NeRF density at all points along the trajectory is below a threshold $\bar{\rho}$ and thus collision-free.

This problem, in its original form without the NeRF, is well-studied in the planning and trajectory optimization community. Its mathematical structure allows it to be formulated as a quadratic programming (QP) problem, which can be solved efficiently. Being able to incorporate computer vision models in the problem formulation expands the scope of research



(a) (b) (c)

Figure 2: NeRF renders of three examples with a collision-free minimum snap optimized trajectory through the NeRF object. In each example, we vary the start point (blue) and goal point (green).

possibilities but can simultaneously introduce greater complexity to the optimization task: The NeRF in (5) makes the resulting NLP highly non-convex due to the non-smooth density landscape as depicted in Fig. 3.

3.2.2. Implementation

We utilize a pre-trained NeRF of a yellow Lego bulldozer, a commonly employed example in NeRF research. IPOPT Wächter and Biegler (2006) is again used as a solver. However, due to the complexity of the problem, we apply a two-stage approach. The first phase optimizes the trajectory to closely follow a sparse set of predetermined collision-free points, excluding the NeRF constraint in Eq. (6d). This results in a sub-optimal yet feasible trajectory. In the second phase, we initialize the solver with the first-phase solution and optimize for the full NLP in Eq. (6) including constraining all points to have a lower NeRF density than $\bar{\rho}=1$. To ensure IPOPT utilizes the warmstarted trajectory in phase two, we initialize IPOPT with a small barrier parameter $\mu=1e^{-4}$ (See Eq. (3a) in Wächter and Biegler (2006)). This compels IPOPT to remain within the region of feasible solutions from the beginning.

3.2.3. Outcome and Visualization

The optimized trajectory for three different configurations with varying start- and goal-points are shown in Fig. 2. For a better understanding of the problem space and the solution trajectories, we also visualize a 2D slice through the environment with the three optimal trajectories in Fig. 3. The optimal minimal snap trajectories are clearly collision-free with the NeRF's density representation. We encourage the reader to view the accompanying animations for this example[4].

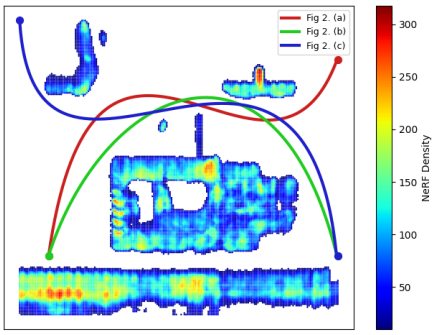


Figure 3: 2D slice of Fig. 2.

4. Further visualizations and animations for the NeRF example can be found at https://github.com/Tim-Salzmann/l4casadi/blob/main/examples/nerf_trajectory_optimization/README.md.

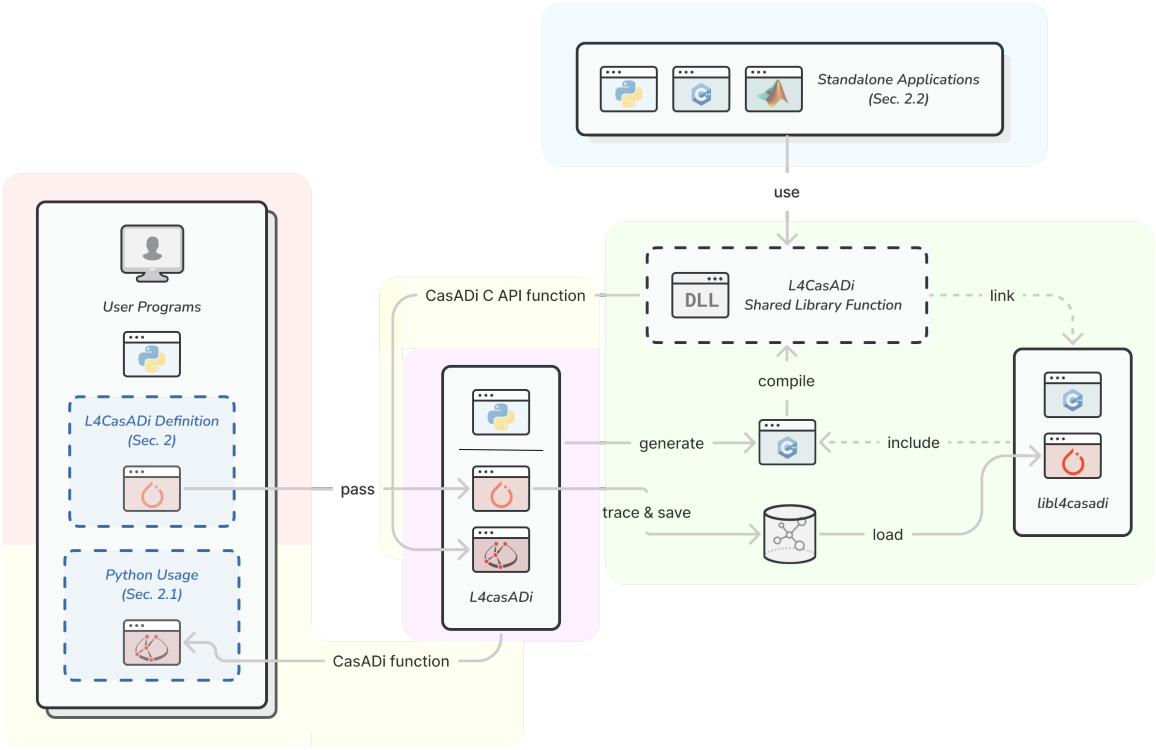


Figure 4: Software Architecture of *L4CasADi*. The block details are presented in Section 4.

4. Framework Architecture

To facilitate a comprehensive grasp of the *L4CasADi* framework's internal operations, the software architecture of the *L4CasADi* framework is presented in Fig. 4. We will guide the reader through the architecture step-by-step, using the figure's background colors for orientation.

Red In accordance with Section 2, *L4CasADi* models are invariably constructed in a Python user program. This entails constructing the PyTorch model and subsequently passing it to the L4CasADi Python class instantiation.

Purple *L4CasADi* symbolically traces the forward pass, the Jacobian, and Hessian of the passed PyTorch model and stores these symbolic representations. Tracing is supported for virtually every PyTorch model architecture. Further, *L4CasADi* auto-generates the function as C++ code.

Green The generated code includes the functionality of the *libl4casadi* library, which facilitates the evaluation of saved PyTorch Traces efficiently within C/C++. These functionalities are exported adhering to the CasADi C interface. The generated code is automatically compiled to a shared library (*L4CasADi* Shared Library Function).

Yellow For direct use of the *L4CasADi* model within Python (as described in Section 2.1) the shared library, exporting function symbols adhering to CasADi's C interface, is loaded

by the Python interface of CasADi within the *L4CasADi* Python class (purple). L4CasADi returns this external function to the Python user program as a CasADi function, which at that point can be seamlessly included in any CasADi computational graph and optimization. By providing derivative information complying with CasADi's C interface naming convention, first and second-order derivatives of the L4CasADi model are available in the CasADi graph.

Blue In addition to directly using the L4CasADi model in the instantiating Python program, the compiled *L4CasADi* Shared Library Function can be loaded and used in C/C++, Matlab, or Python standalone applications (See Section 2.2).

5. Related Work

The increasing prevalence of data-driven models in optimization has spurred a surge of scholarly contributions, making it infeasible to comprehensively review the entire landscape of this domain within the scope of this discussion. Instead, we will concentrate on specific frameworks that integrate learned models with numerical optimization techniques, specifically PyTorch and CasADi.

Learning and Numerical Optimization Multiple approaches have been proposed to bring advanced numerical optimization algorithms within learning frameworks like PyTorch and TensorFlow Amos et al. (2018); Wang et al. (2023). However, these integrations often result in the re-implementation of individual solver algorithms within the learning framework, which may not exhibit the same robustness and maturity as the well-established solvers within CasADi. Recently, NeuroMANCER Drgona et al. (2023) enables the formulation of optimization problems entirely in PyTorch. Naturally, learned components in PyTorch can be included, even optimized within the problem solution. However, because NeuroMANCER is restricted to the PyTorch environment they are mostly limited to the first-order solvers within — not using second-order approaches such as IPOPT or SQP which have been proven to be efficient and robust.

PyTorch and CasADi The demand for such a framework within the community is evident from prior endeavors aimed at uniting the two concepts of learning in PyTorch and optimization in CasADi. Multiple packages, *do-mpc* Fiedler et al. (2023), *HILO-MPC* Pohlodek et al. (2022), and *ML-CasADi* Salzmann et al. (2023), provide the capabilities to rebuild simple architectures of PyTorch models directly in CasADi by copying the learned weight tensors and formulate matrix multiplications and activation functions in CasADi. This approach, similar to Naïve *L4CasADi* (Section 2.3), is restricted to models comprised of the limited CasADi function set. Further, CasADi's function set is not optimized for large matrix multiplications and can not use hardware acceleration. Additionally, Salzmann et al. (2023); Lahr et al. (2023) calculate function evaluations and sensitivities for the learned model separated from CasADi in the PyTorch framework and subsequently injects these results into the CasADi graph. The applicability of this approach, however, is limited to specific optimization algorithms and introduces inefficiencies in the form of excessive context switches and memory transfers between CasADi and PyTorch.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 3 2016.

Brandon Amos, Ivan Dario Jimenez Rodriguez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable MPC for End-to-End Planning and Control. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 8299–8310, Red Hook, NY, USA, 2018. Curran Associates Inc.

Joel A. E. Andersson, Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl. CasADi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1):1–36, 3 2019. ISSN 1867-2949. doi: 10.1007/s12532-018-0139-4.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

Michael L Bynum, Gabriel A Hackebeil, William E Hart, Carl D Laird, Bethany L Nicholson, John D Siirola, Jean-Paul Watson, and David L Woodruff. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, third edition, 2021.

Steven Diamond and Stephen Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *J. Mach. Learn. Res.*, 17(1):2909–2913, 1 2016. ISSN 1532-4435.

Alexander Domahidi, Eric Chu, and Stephen Boyd. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*, pages 3071–3076. IEEE, 7 2013. ISBN 978-3-033-03962-9. doi: 10.23919/ECC.2013.6669541.

Jan Drgona, Aaron Tuor, James Koch, Madelyn Shapiro, and Draguna Vrabie. NeuroMANCER: Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations. 2023.

Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. qpOASES: a parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, 12 2014. ISSN 1867-2949. doi: 10.1007/s12532-014-0071-1.

Felix Fiedler, Benjamin Karg, Lukas Lüken, Dean Brandner, Moritz Heinlein, Felix Brabender, and Sergio Lucia. do-mpc: Towards FAIR nonlinear and robust model predictive control. *Control Engineering Practice*, 140:105676, 11 2023. ISSN 09670661. doi: 10.1016/j.conengprac.2023.105676.

Robert Fourer, David M. Gay, and Brian W. Kernighan. AMPL: A Mathematical Programing Language. In *Algorithms and Model Formulations in Mathematical Programming*, pages 150–151. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989. doi: 10.1007/978-3-642-83724-1.

Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM Review*, 47(1):99–131, 1 2005. ISSN 0036-1445. doi: 10.1137/S0036144504446096.

Gurobi Optimization LLC. Gurobi Optimizer Reference Manual, 2023.

Amon Lahr, Andrea Zanelli, Andrea Carron, and Melanie N. Zeilinger. Zero-order optimization for Gaussian process-based model predictive control. *European Journal of Control*, 74, 11 2023. ISSN 09473580. doi: 10.1016/j.ejcon.2023.100862.

MOSEK ApS. The MOSEK optimization toolbox., 2023.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 12 2019.

Johannes Pohlodek, Bruno Morabito, Christian Schlauch, Pablo Zometa, and Rolf Findeisen. Flexible development and evaluation of machine-learning-supported optimal control and estimation methods via HILO-MPC, 2022.

Tim Salzmann, Elia Kaufmann, Jon Arrizabalaga, Marco Pavone, Davide Scaramuzza, and Markus Ryll. Real-Time Neural MPC: Deep Learning Model Predictive Control for Quadrotors and Agile Robotic Platforms. *IEEE Robotics and Automation Letters*, 8(4): 2397–2404, 4 2023. ISSN 2377-3766. doi: 10.1109/LRA.2023.3246839.

Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 12 2020. ISSN 1867-2949. doi: 10.1007/s12532-020-00179-2.

Robin Verschueren, Gianluca Frison, Dimitris Kouzoupis, Jonathan Frey, Niels van Duijkeren, Andrea Zanelli, Branimir Novoselnik, Thivaharan Albin, Rien Quirynen, and Moritz Diehl. acados: a modular open-source framework for fast embedded optimal control. *Mathematical Programming Computation*, 10 2021.

Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 3 2006. ISSN 0025-5610. doi: 10.1007/s10107-004-0559-y.

Wenxuan Wang, Yuhang Zhang, Jiaxin Gao, Yuxuan Jiang, Yujie Yang, Zhilong Zheng, Wenjun Zou, Jie Li, Congsheng Zhang, Wenhan Cao, Genjin Xie, Jingliang Duan, and Shengbo Eben Li. GOPS: A general optimal control problem solver for autonomous driving and industrial control applications. *Communications in Transportation Research*, 3:100096, 12 2023. ISSN 27724247. doi: 10.1016/j.commtr.2023.100096.