PyPose v0.6: The Imperative Programming Interface for Robotics

Zitong Zhan[1], Xiangfu Li[2], Qihang Li[1], Haonan He[3], Abhinav Pandey[2], Haitao Xiao[4], Yangmengfei Xu[5], Xiangyu Chen[1], Kuan Xu[6], Kun Cao[6], Zhipeng Zhao[1], Zihan Wang[3], Huan Xu[7], Zihang Fang[8], Yutian Chen[3], Wentao Wang[2], Xu Fang[6], Yi Du[1], Tianhao Wu[3], Xiao Lin[7], Yuheng Qiu[3], Fan Yang[9], Jingnan Shi[10], Shaoshu Su[1], Yiren Lu[1], Taimeng Fu[1], Karthik Dantu[1], Jiajun Wu[11], Lihua Xie[6], Marco Hutter[9], Luca Carlone[10], Sebastian Scherer[3], Daning Huang[2], Yaoyu Hu[3], Junyi Geng[2], Chen Wang[1]

https://pypose.org

***Abstract*— PyPose is an open-source library for robot learning. It combines a learning-based approach with physics-based optimization, which enables seamless end-to-end robot learning. It has been used in many tasks due to its meticulously designed application programming interface (API) and efficient implementation. From its initial launch in early 2022, PyPose has experienced significant enhancements, incorporating a wide variety of new features into its platform. To satisfy the growing demand for understanding and utilizing the library and reduce the learning curve of new users, we present the fundamental design principle of the imperative programming interface, and showcase the flexible usage of diverse functionalities and modules using an extremely simple Dubins car example. We also demonstrate that the PyPose can be easily used to navigate a real quadruped robot with a few lines of code.**

I. Introduction

PyPose is a Python-based, robotics-oriented, and open-source library designed for researchers and rapid prototyping [1]. Thanks to its adeptly crafted imperative programming approach, it offers swift customization for distinct applications. PyPose has found use in diverse robotic applications, including odometry [2], control [3], and planning [4]. It aims to bridge the gap in robotic systems where deep learning-based methods and physics-based optimization often end up residing in separate modules implemented by different libraries, which can lead to suboptimal solutions.

Since the last major release (v0.3[†]), PyPose has experienced significant enhancements, incorporating a wide variety of new features into its platform. We have noticed that there is a growing demand for understanding and utilizing PyPose, especially regarding the rationale behind the application programming interface (API) design. To reduce the learning curve of new users, this paper seeks to present the principle of the imperative interface by showcasing several examples covering diverse aspects of robotics, such as state estimation, planning, and control. The contributions of this paper include

- We present the design philosophy behind the imperative interface of the PyPose library. That is, the dynamic system class is a unified API for various functionalities such as state estimation, trajectory smoothing, and control. We show that each function can be done within a few lines of code, using a simple Dubins car example.
- We demonstrate that the control loop can be extended to real-world robots seamlessly, thereby enhancing robots with the ability to utilize various out-of-the-box functionalities within PyPose.

Corresponding Email: admin@pypose.org
[1]State University of New York at Buffalo, Buffalo, NY 14260, USA.
[2]Pennsylvania State University, University Park, PA, 16802, USA.
[3]Carnegie Mellon University, Pittsburgh, PA 15213, USA.
[4]ZBL Co., Ltd., China.
[5]University of Melbourne, Parkville VIC 3052, Australia.
[6]Nanyang Technological University, Singapore 639798.
[7]Georgia Institute of Technology, Atlanta, GA 30332, USA.
[8]Northview High School, Johns Creek, GA 30097, USA.
[9]ETH Zürich, 8092 Zürich, Switzerland.
[10]Massachusetts Institute of Technology, Cambridge, MA 02139.
[11]Stanford University, Stanford, CA 94305, USA.

[†]A history of PyPose is at https://github.com/pypose/pypose/releases

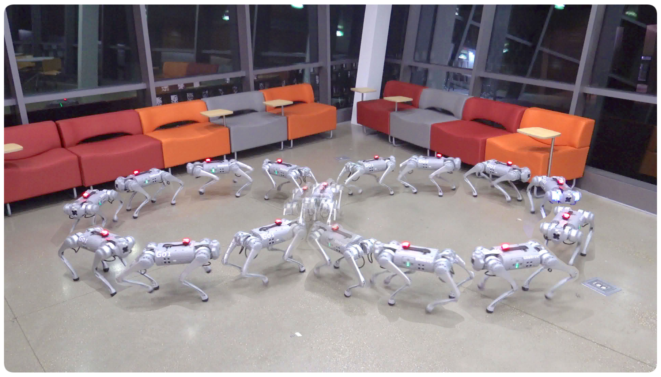


Fig. 1: With just a few lines of code in Python using PyPose, a quadruped robot can trace an ∞-shaped path.

The name "PyPose" was inspired by an intriguing observation: the field of robotics can primarily be divided into four areas: perception, control, planning, and simultaneous localization and mapping (SLAM). Essentially, robot control is about stabilizing its current pose, planning focuses on generating future poses, and SLAM deals with estimating the past and current poses. To differentiate the library from PyTorch [5], which excels in perception, and to emphasize its Python-centric approach and unique focus on robotics, the library was named "PyPose".

II. Related Work

A comprehensive review was conducted in [1]. In this study, we will compare some optimization libraries that are widely used in robotic applications. Ipopt [6] is a C++-based solver targeting non-linear programming. CasADi [7] is a non-linear optimization tool and implements forward mode algorithmic differentiation through a symbolic interface. It serves as a base tool for many software in planning and control. However, both Ipopt and CasADi are not designed to work with robot learning methods. While HILO-MPC [8] based on CasADi attempts to integrate machine learning with optimal control models, it does not provide easy-to-use native PyTorch support, and the control loop needs to be explicitly defined. NeuroMANCER [9] is an open-source library that

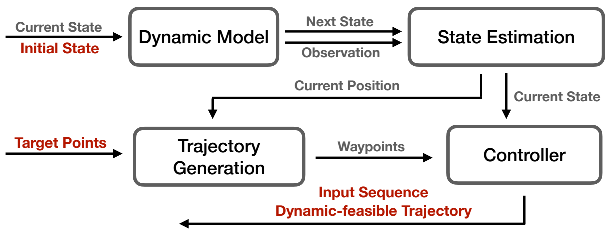


Fig. 2: The operational workflow of the Dubins car system.

leverages PyTorch for differentiable programming, focusing on constrained optimization, dynamic system formulation, and parametric model-based optimal control. To the best of our knowledge, PyPose is a pioneering PyTorch-based library offering a comprehensive interface for robotics such as perception, SLAM, and control involving optimization.

III. Imperative Programming Interface

Aside from the logical and modular design philosophies of PyPose, our principle of defining modules is to keep a general imperative interface that is compatible with a number of accessories. This section elucidates our design principles by focusing on four core modules: Dynamic Systems, State Estimation, Trajectory Interpolation, and Control. To illustrate the seamless integration and utility of these modules, we present a case study where a Dubins car is programmed to navigate an ∞-like trajectory. The operational workflow of the entire system is depicted in Fig. 2.

A. Dynamic System

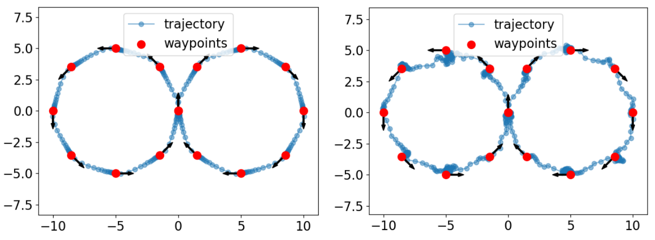
Dynamics provides a framework for describing the behavior of systems as they evolve over time. Such behavior can be generally modeled through two fundamental equations representing prediction $\mathbf{f}$ and observation $\mathbf{g}$:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, t_k) + \mathbf{w}_k \tag{1a}$$
$$\mathbf{y}_k = \mathbf{g}(\mathbf{x}_k, \mathbf{u}_k, t_k) + \mathbf{v}_k \tag{1b}$$

where t_k, $\mathbf{x}_k \in \mathbb{R}^N$, $\mathbf{u}_k \in \mathbb{R}^C$, $\mathbf{y}_k \in \mathbb{R}^M$ are the system time, states, inputs, and observations at the k-th time step, respectively. The terms $\mathbf{w}$ and $\mathbf{v}$ represent noise in the state transition and observation functions, respectively.

We design `System` as a parent class for all dynamics systems including linear time-invariant (`LTI`) [10], linear time-variant (`LTV`) [11], and non-linear system (`NLS`) [12]. They not only carry the core functionality of performing state transition and system observation, but also provide the interface enabling users to access the key properties, such as linearized system matrices (automatically used by nonlinear control modules), etc. `System` module and its sub-classes feature two user-defined class methods `state_transition` and `observation`. To utilize those dynamic modeling capabilities, users simply need to subclass one of the predefined dynamic system templates and implement their specific state transition and observation methods according to their needs.

With this architecture, the inherited `forward` method from `torch.nn.Module` is used to handle state transitions for discrete-time systems and advances the time step.



(a) DubinCar trajectory in ideal case (b) DubinCar trajectory with noise

Fig. 3: `DubinCar` system following waypoints.

This design streamlines the implementation of discrete-time systems, which are commonly used in robotics.

We next demonstrate the simplicity of defining a system with a `DubinsCar` model in (2). Generally, the state is defined as a position and heading in 2D plane and the inputs $\mathbf{u} = [v, \varphi]$ are linear velocity and angular rate

$$\theta_{k+1} = \theta_k + \varphi \cdot \Delta t \tag{2a}$$
$$i_{k+1} = i_k + v \cdot \cos\theta_{k+1} \cdot \Delta t \tag{2b}$$
$$j_{k+1} = j_k + v \cdot \sin\theta_{k+1} \cdot \Delta t \tag{2c}$$

In practical terms, to avoid the complications of angle wrapping, we opt for using the trigonometric values $\cos(\theta)$ and $\sin(\theta)$ instead of θ itself, and the system state is in the form of $[i, j, \cos(\theta), \sin(\theta)]$. This approach mitigates the issues of periodicity and discontinuity in angular values.

Due to its non-linearity, the `DubinsCar` model can be defined as a subclass of `NLS`, where the users only need to define methods `state_transition` and `observation`.

```python
class DubinsCar(pp.module.NLS):
    # A 2-D DubinsCar kinematic model.
    def __init__(self, dt=0.1):
        super().__init__()
        self.dt = dt

    def state_transition(self, x, u, t=None):
        v, phi = u[..., 0], u[..., 1]
        c, s = x[..., 2], x[..., 3]
        theta = torch.atan2(s, c) + phi * dt

        i = x[..., 0] + v * c * dt
        j = x[..., 1] + v * s * dt
        c, s = theta.cos(), theta.sin()
        return torch.stack([i, j, c, s], dim=-1)

    def observation(self, x, u, t):
        return x
```

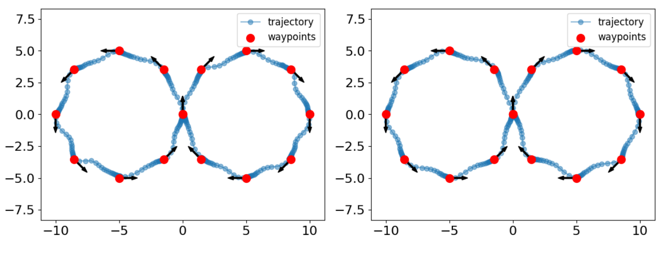
In this example, we override $\mathbf{f}$ in `state_transition` and $\mathbf{g}$ in `observation`. The model takes the current state `x` and an input `u[i]` as arguments, advances one time step, and returns the next state and observation.

```python
car = DubinsCar()
for i in range(N - 1):
    x[i+1], y[i] = car(x[i], u[i])
```

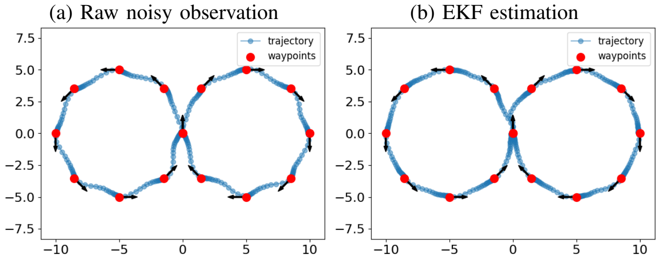
Fig. 3a shows an ideal `DubinsCar` model follows the ∞-trajectory without noise using an optimal input sequence calculated from the next section.

B. Optimal Control Solvers

We next show that an optimal controller can be calculated on the fly based on the model's real-time state in one



(a) Raw noisy observation (b) EKF estimation



(c) UKF estimation (d) PF estimation

Fig. 4: MPC relying in different filters

line of code. PyPose provides several differentiable optimal controllers, including Linear Quadratic Regulator (`LQR`) [13] and Model Predictive Control (`MPC`) [14]. LQR in PyPose can be used for both linear time-invariant (`LTI`) and linear time-varying (`LTV`) systems. Below we provide an example of MPC, controlling the `DubinsCar` model tracing way-points on an ∞-shaped trajectory. As illustrated in Fig. 3b, the Dubins car visits each of the red waypoints, set as intermediate goal state, in order. Its trajectory starts and ends at the center $(0, 0)$ location. The code snippet demonstrates MPC moving the Dubins car to traverse through all of the 17 waypoints (the start, end, and intermediate points overlap at the center location). Given the `target` parameter, the `MPC` computes the optimum input and iteratively moves the `DubinsCar` model until the current red target position is reached. Each blue connected dot represents the updated Dubins car state in one iteration.

```
mpc = pp.module.MPC(car, Q, p, T)
```

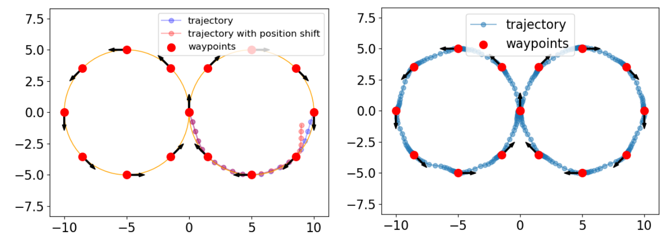
where `Q` and `p` represent the weight matrices for the quadratic terms and the weight vectors for the linear terms at each time step, respectively. `T` is the time horizon on which MPC solves the optimization. Then an MPC object can be used as

```
v = q * torch.randn(4) # observation noise
xt, u, cost = mpc(dt, y[i] + v, target)
```

where `dt` is the time step interval, `y[i]` is the observation from the previous iteration, `v` is the injected noice, and `target` is the goal state defined in the exact same format as the `DubinsCar` model's state; then the expected states `xt`, system inputs `u`, and `cost` along the time horizon will be outputted. This example shows that the controller manages the robot's path effectively but the trajectory fluctuates in the presence of noise. We next show that a state estimation module can be applied to further reduce the effect of noises.

C. State Estimation

PyPose provided the commonly used Bayesian filters [15] including extend Kalman filter (`EKF`) [16], unscented Kalman filter (`UKF`) [17], and particle filter (`PF`) [18]. Below we demonstrate the usage of `EKF` on the same `DubinsCar` system, and we hereby show that its usage only requires two additional lines of code. The first line of code defines the EKF module by wrapping up the Dubins car model.



(a) Local planned trajectory (b) Trajectory with EKF and spline

Fig. 5: `DubinCar` system with `chspline`

```
ekf = pp.module.EKF(car)
```

Each time the system is propagated, the filter module takes the same format of input as the dynamics module. The second line of code is called immediately after the system is propagated with an observation produced.

```
X[i+1], P = ekf(X[i], y[i], u[i], P, Q, R)
```

where `X[i]` is the estimated state from the previous step and `y[i]` is the noisy observation; `P`, `Q`, and `R` are the state estimation covariance of the previous step, covariance of system transition model, and covariance of system observation model, respectively. The estimated observation could be used in MPC to generate more accurate control signals when observation noise is present. Fig. 4 shows results produced by MPC when it takes either the raw noisy observation or the estimation from filters as input to follow the same trajectory.

D. Trajectory Interpolation

PyPose provides trajectory interpolation algorithms, B-spline (`bspline`) [19] and Cubic-Hermite Spline (`chspline`) [20] as the guidance for trajectory smoothing. The trajectory interpolation module serves as a valuable asset for robotic planning in uncertain environments, the dense smooth waypoints generated could guide the robot with a position shift back to the desired trajectory. Furthermore, the dense waypoints could decrease the computing cost of MPC and overshoot probability. The `chspline` function in PyPose is user-friendly and straightforward to implement. By simply inputting a sequence of waypoints, the function generates an evenly distributed, smooth trajectory with a continuous first derivative. The code snippet below provides an example of how to employ the `chspline` function for online local trajectory planning with the `DubinsCar` model.

```
points = pp.chspline(waypoints, interval=0.2)
```

Fig. 5a provides an example of local planning using the `spline` algorithm. The orange line serves as the ground truth trajectory, represented as two circles. The black arrows on the waypoints indicate the trajectory direction. The blue trajectory is generated in accordance with the planned path, while the red trajectory emerges when the robot deviates

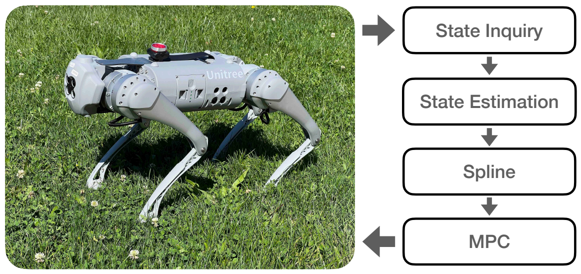


Fig. 6: Flow chart for the quadruped robot experiment.

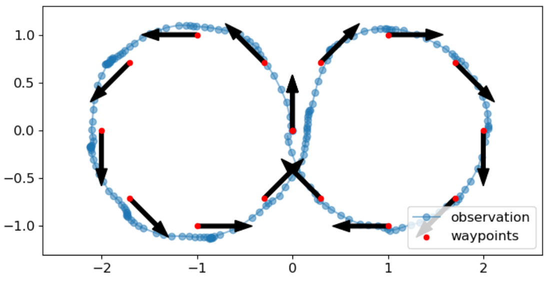


Fig. 7: The trajectory recorded on the quadruped robot in Fig. 1.

from this path due to an initial positional shift. Notably, the blue trajectory exhibits a high degree of alignment with the ground truth, attesting to the module's capability to generate optimal trajectories. Moreover, the red trajectory eventually merges smoothly with the blue one, illustrating the module's ability to produce smooth trajectories even when positional shifts occur with the Dubins car.

Fig. 5b depicts a trajectory generated by combining the `EKF` and `chspline` algorithms. When compared to the results solely from `EKF` as shown in Fig. 4b, the inclusion of `chspline` noticeably smooths the trajectory and brings it closer to the desired ∞-shaped path. The algorithms not only generate optimal and smooth trajectories but also demonstrate resilience in accommodating positional shifts. These methods also support batch processing of waypoint sequences, enabling the generation of multiple trajectories simultaneously. This feature enhances computational efficiency and expedites the planning process.

IV. Experiments

This experiment aims to showcase the real-world applicability of PyPose by integrating it with a Unitree Go1 quadruped robot. The task involves adding the execution on a real robot into the PyPose implemented control loop, as indicated in Fig. 6, and navigating the robot through the waypoints on the ∞-shaped trajectory. The assumption is that the robot has built-in localization producing reasonably accurate position estimation, and can be controlled through high-level motion commands. The controller on the robot is a cluster of single-board computers loaded with a Ubuntu desktop. The control loop could be executed on any remote computer within the same local network as the robot.

In each iteration of the control loop, it inquires about the position and heading of the robot. Using the inquired pose as observation, the control loop sends a command to the robot based on optimal control from MPC, until the robot reaches a target. Finally, the trajectory illustrated in Fig. 7 is recorded after the robot traverses through each of the target locations. Each of the blue connected dots represents an observation at a time step. Stacked snapshots of the real environment during execution are shown in Fig. 1. This indicates that PyPose can effectively guide the robot along the desired path.

V. Conclusions & Discussion

In this paper, we highlighted the conciseness of the PyPose library's imperative interface. It offers a unified API for various functionalities, all achievable with a few lines of code. Our experiments showed that PyPose integrates seamlessly with a quadruped robot for navigation and equips robots with a myriad of ready-to-use features. It is worth noting that all functions presented in this paper are fully differentiable, which makes an attractive starting point for the development of more advanced end-to-end robot learning systems. We expect that PyPose will inspire broader robotics research.

References

[1] C. Wang, D. Gao, K. Xu, J. Geng, Y. Hu, Y. Qiu, B. Li, F. Yang, B. Moon, A. Pandey, Aryan, J. Xu, T. Wu, H. He, D. Huang, Z. Ren, S. Zhao, T. Fu, P. Reddy, X. Lin, W. Wang, J. Shi, R. Talak, K. Cao, Y. Du, H. Wang, H. Yu, S. Wang, S. Chen, A. Kashyap, R. Bandaru, K. Dantu, J. Wu, L. Xie, L. Carlone, M. Hutter, and S. Scherer, "PyPose: A library for robot learning with physics-based optimization," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023. [Online]. Available: https://arxiv.org/pdf/2209.15428.pdf

[2] T. Fu, S. Su, and C. Wang, "iSLAM: Imperative SLAM," *arXiv preprint arXiv:2306.07894*, 2023. [Online]. Available: https://arxiv.org/pdf/2306.07894.pdf

[3] A. Pandey, D. Huang, Y. Yu, and J. Geng, "Learning koopman operators with control using bi-level optimization," in *2023 IEEE 62st Conference on Decision and Control (CDC)*. IEEE, 2023.

[4] F. Yang, C. Wang, C. Cadena, and M. Hutter, "iplanner: Imperative path planning," in *Robotics: Science and Systems (RSS)*, 2023. [Online]. Available: https://arxiv.org/pdf/2302.11434.pdf

[5] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[6] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, pp. 25–57, 2006.

[7] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, 2018.

[8] J. Pohlodek, B. Morabito, C. Schlauch, P. Zometa, and R. Findeisen, "Flexible development and evaluation of machine-learning-supported optimal control and estimation methods via hilo-mpc," 2022.

[9] A. Tuor, J. Drgona, J. Koch, M. Shapiro, D. Vrabie, and S. Briney, "NeuroMANCER: Neural Modules with Adaptive Nonlinear Constraints and Efficient Regularizations," 2023.

[10] https://pypose.org/docs/main/generated/pypose.module.LTI/.

[11] https://pypose.org/docs/main/generated/pypose.module.LTV/.

[12] https://pypose.org/docs/main/generated/pypose.module.NLS/.

[13] https://pypose.org/docs/main/generated/pypose.module.LQR/.

[14] https://pypose.org/docs/main/generated/pypose.module.MPC/.

[15] D. Simon, *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[16] https://pypose.org/docs/main/generated/pypose.module.EKF/.

[17] https://pypose.org/docs/main/generated/pypose.module.UKF/.

[18] https://pypose.org/docs/main/generated/pypose.module.PF/.

[19] https://pypose.org/docs/main/generated/pypose.bspline/.

[20] https://pypose.org/docs/main/generated/pypose.chspline/.