




The o80 C++ templated toolbox: Designing customized Python APIs for synchronizing realtime processes

Vincent Berenz^{*1}, Maximilien Naveau², Felix Widmaier³, Manuel Wüthrich³, Jean-Claude Passy⁴, Simon Guist³, and Dieter Büchler³

¹ Autonomous Motion, Max Planck Institute for Intelligent Systems ² Movement Generation and Control, Max Planck Institute for Intelligent Systems ³ Empirical Inference, Max Planck Institute for Intelligent Systems ⁴ Software Workshop, Max Planck Institute for Intelligent Systems

DOI: [10.21105/joss.02459](https://doi.org/10.21105/joss.02459)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Pending Editor](#) 

Submitted: 07 July 2020

Published: 10 July 2020

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

Statement of need

o80 (pronounced “oh-eighty”) is a software for synchronizing and organizing message exchange between (realtime) processes via simple customized Python APIs. Its target domain is robotics and machine learning. Our motivation for developing o80 is to ease the setup of robotics experiments (i.e. integration of various hardware and software) by machine learning scientists. Such setup typically requires time and technical effort, especially when realtime processes are involved. Ideally, scientists should have access to a simple Python API that hides the lower level communication details and simply allows to send actions and receive observations. o80 is a tool box for creating such API.

Overview

For implementing synchronization, o80 organizes two types of processes:

- A server process encapsulates an instance of o80 back-end and an instance of a driver. At each iteration, the back-end instance computes for each actuator the desired state to be transmitted to the hardware via the driver. The back-end also reads sensory information from the driver. Typically, the server process is programmed in C++ with consideration for realtime.
- A client process encapsulates instances of o80 front-end, which provides: 1) an interface to send commands to the back-end (i.e. requests to compute desired states), 2) methods for querying sensory information, and 3) methods for synchronizing the client with the server process.

In the background, back-end and front-end(s) communicate by exchanging serialized object instances via a interprocess shared memory. Serialization is based on the cereal library (Shane & Randolph, 2017), and the shared memory is based on the Boost library (Schling, 2011).

o80 is templated over actuator states and driver; and may therefore support a wide range of systems.

o80's core API supports:

- methods for specifying via commands either full desired state trajectories or partial trajectories relying on interpolation.

*corresponding author

- interpolation methods based on specified duration, speed, or number of server iterations.
- methods for either queuing or interrupting trajectories.
- frontend methods for setting commands that are either blocking or non blocking.
- frontend methods for retrieving sensory data that request the latest available information, sensor information corresponding to a specific past server iteration, or the full history of sensory information.
- client processes and the server processes that run asynchronously or synchronously.

Synchronization methods may be based on a fixed desired frequency set for the server, or may be set up by the client process (“bursting mode”).

o80 library may be considered complex as it is versatile and can be used in the context of a wide range of control strategies. Yet, the objective of o80 is to provide robot users with a simple Python API. For this purpose, o80 provides convenience functions for generating application tailored Python bindings. The expected usage is that an expert developer uses o80 to design the Python API that will hide to end users the complexities related to interprocess communication and synchronization. Scientists may then use this simplified API to design new robotic experiments. In this sense, o80 aims to be a toolbox for creating customized Python APIs. Generation of Python bindings via the o80 API is based on pybind11 (Jakob, Rhineland, & Moldovan, 2016).

Modular Implementation

o80 is based on open source packages that are maintained by the Max Planck Institute for Intelligent Systems, and which may be reused in other contexts. These packages are available on the GitHub domains “intelligent soft robots”, “machines in motion”, “mpi-is” and “open dynamic robot initiative”. Examples of such packages are:

- synchronizer: a library for synchronizing processes
- shared memory: a wrapper over the boost interprocess library that makes exchange of serialized data over an interprocess shared memory trivial
- time series: a templated circular buffer with time stamps, supporting multiprocess access and synchronization

Examples of usage

Integration with SL

An instance of o80 backend has been integrated into the SL realtime library (Schaal, 2009) used for the control of the Apollo manipulator (Kappler et al., 2018). This allows scientists to program robot behavior using a simple Python interface running at a low non-realtime frequency that synchronizes with the realtime higher frequency C++ control loop of the robot. This interface was used for example for the experiments described in (Baumann, Solowjow, Johansson, & Trimpe, 2020).

HYSR training

o80 has been used in the context of reinforcement learning applied to real robotic hardware. (Büchler et al., 2020) describes a setup in which a robot arm driven by pneumatic artificial muscles learns autonomously to play table tennis using an hybrid sim and real training approach

(HYSR), i.e., performing real hardware motions to interact with simulated balls. o80 was used in this context to:

- provide a Python API that has been integrated into a Gym environment to provide an interface to reinforcement algorithms (Brockman et al., 2016)
- setting up the synchronization between the real robot control and the MuJoCo simulator (Todorov, Erez, & Tassa, 2012) used for HYSR
- setting up asynchronous processes for live plotting of the robot state

References

- Baumann, D., Solowjow, F., Johansson, K. H., & Trimpe, S. (2020). Identifying causal structure in dynamical systems. Retrieved from <http://arxiv.org/abs/2006.03906>
- Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI gym. Retrieved from <http://arxiv.org/abs/1606.01540>
- Büchler, D., Guist, S., Calandra, R., Berenz, V., Schölkopf, B., & Peters, J. (2020). Learning to play table tennis from scratch using muscular robots. Retrieved from <http://arxiv.org/abs/2006.05935>
- Jakob, W., Rhineland, J., & Moldovan, D. (2016). Pybind11 — seamless operability between c++11 and python. Retrieved from <https://github.com/pybind/pybind11>
- Kappler, D., Meier, F., Issac, J., Mainprice, J., Garcia Cifuentes, C., Wüthrich, M., Berenz, V., et al. (2018). Real-time perception meets reactive motion generation. *IEEE Robotics and Automation Letters*, 3(3), 1864–1871. doi:[10.1109/LRA.2018.2795645](https://doi.org/10.1109/LRA.2018.2795645)
- Schaal, S. (2009). The sl simulation and real-time control software package. Retrieved from <http://www-clmc.usc.edu/publications/S/schaal-TRSL.pdf>
- Schling, B. (2011). *The boost c++ libraries*. XML Press. ISBN: [0982219199](https://www.amazon.com/dp/0982219199)
- Shane, G. W., & Randolph, V. (2017). Cereal - a c++11 library for serialization. Retrieved from <http://uscilab.github.io/cereal/>
- Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *IROS* (pp. 5026–5033). IEEE. doi:[10.1109/iros.2012.6386109](https://doi.org/10.1109/iros.2012.6386109)