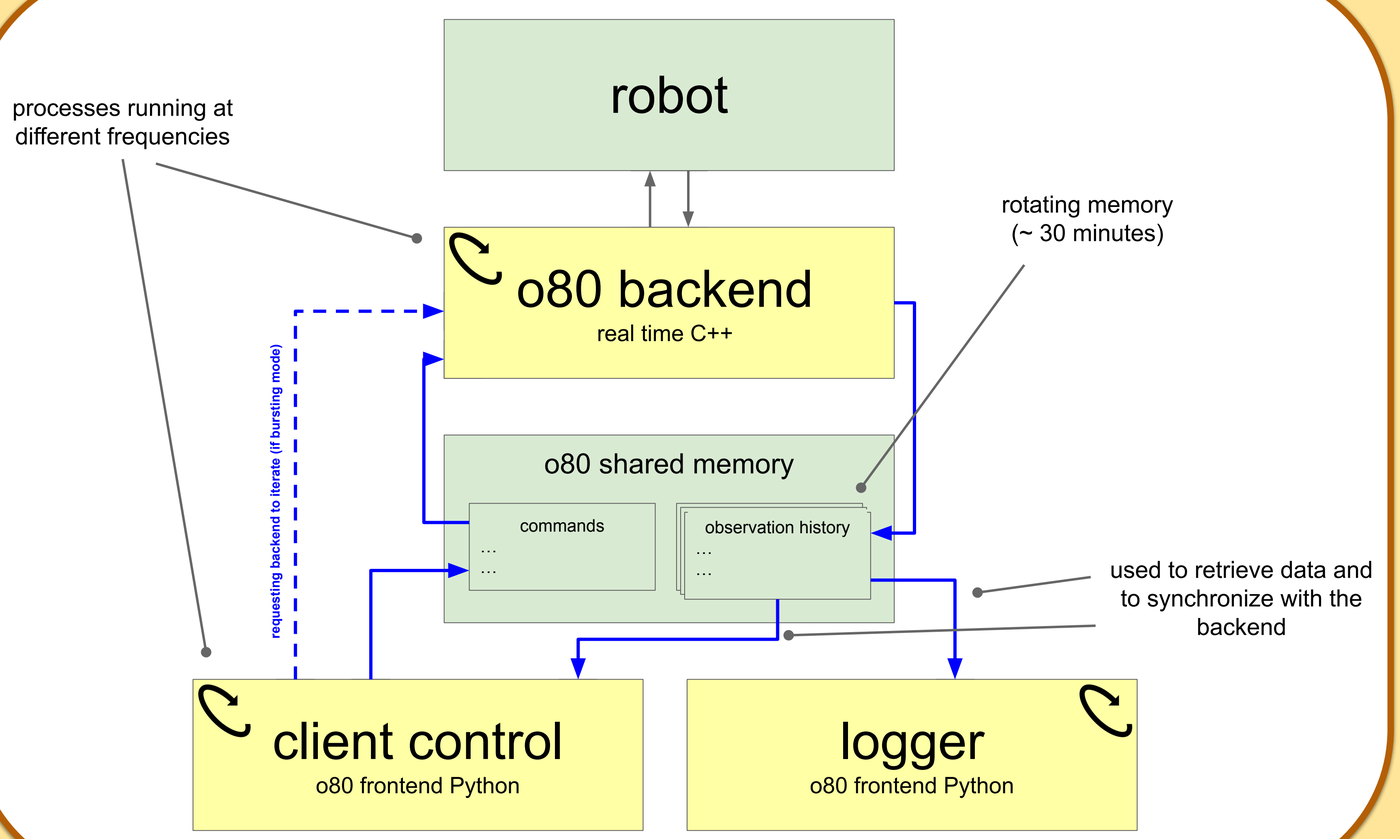
Synchronizing Machine Learning Algorithms, Realtime Robotic Control and Simulated Environment with o80

Vincent Berenz, Felix Widmaier, Simon Guist, Bernhard Schölkopf and Dieter Büchler

o80 is a software for:

- managing inter-processes data streams exchange
- inter-processes synchronization

It is templated realtime safe C++, with automated generation of Python bindings.
Open source:
https://github.com/intelligent-soft-robots/o80



Deployment

1. develop the classes for the driver (specifies input and output to robot) and the joints (information related to the robot state)

```
class Standalone
  : public o80::Standalone<40000,  // shared memory history size
                           4,     // nb of dofs
                           Driver, // CUSTOM DRIVER CLASS
                           Joint,  // CUSTOM JOINT CLASS
                           >
```

2. create the Python bindings at compile time

```
PYBIND11_MODULE(o80_robot, m)
{
    o80::create_python_bindings<Standalone>(m);
    o80::create_standalone_python_bindings<Driver,
                                           Standalone>(m)
}
```

3. start the (realtime safe) o80 backend in Python

```
start_standalone(SEGMENT_ID, FREQUENCY, BURSTING_MODE)
```

4. interact with the backend via a frontend (in Python)

```
frontend = FrontEnd(SEGMENT_ID)
observation = frontend.pulse()
```

Flexible command system

```
dof = 1

frontend.add_command(dof, 100, o80.Duration.milliseconds(2000), o80.mode.OVERWRITE)

frontend.add_command(dof, 200, o80.Speed.per_milliseconds(10), o80.mode.QUEUE)

frontend.add_command(dof, 300, o80.Iteration(10000), o80.mode.QUEUE)

frontend.add_command(dof, 301, o80.mode.QUEUE)

frontend.pulse()
```

queuing a command requesting the degree of freedom 1 to reach the value 100 over 2 seconds, overwriting any previously running commands

… requesting to then reach the value 200 at 10 units per millisecond

… requesting to then reach the value 300 at the 10000th iteration of the backend

… requesting to then reach asap the value 301

writing the queue of commands to the shared memory for execution

Reading data and synchronizing

```
# reading the latest observation
observation = frontend.latest()
iteration = observation.get_iteration()

# reading an observation from the past
observation = frontend.read(iteration - 1000)

# reading the 100 latest observations
observations = frontend.get_latest_observations(100)

# reading the latest observations
observations = frontend.get_observations_since(iteration)

# waiting for future observations !
# -> synchronizing with backend
observation = frontend.read(iteration + 1000)
observation = frontend.wait_for_next()
```

flexible methods for retrieving observations and synchronizing with the backend

```
observation = frontend.latest()
iteration = observation.get_iteration()

while True:
    iteration += 10
    observation = frontend.read(iteration + 10)
    # creating and queuing commands
    frontend.pulse()
```

example:
the frontend runs at 1/10th of the backend's frequency

Bursting mode

In bursting mode, the backend iterates upon requests by the frontend.
Typically this is used for the control of simulated robot running accelerated time.
This allows the simulation to synchronize with the control algorithm.

example:
the control algorithm computes the commands to be sent, then request the simulation to perform 10 iterations as fast as possible

```
while True:
    observation = frontend.latest()
    # creating and queuing commands
    frontend.burst(10)
```

https://ei.is.mpg.de

EMPIRICAL INFERENCE
MAX PLANCK INSTITUTE FOR INTELLIGENT SYSTEMS