taskontrol Documentation

Release 1.0

Santiago Jaramillo

Dec 30, 2020

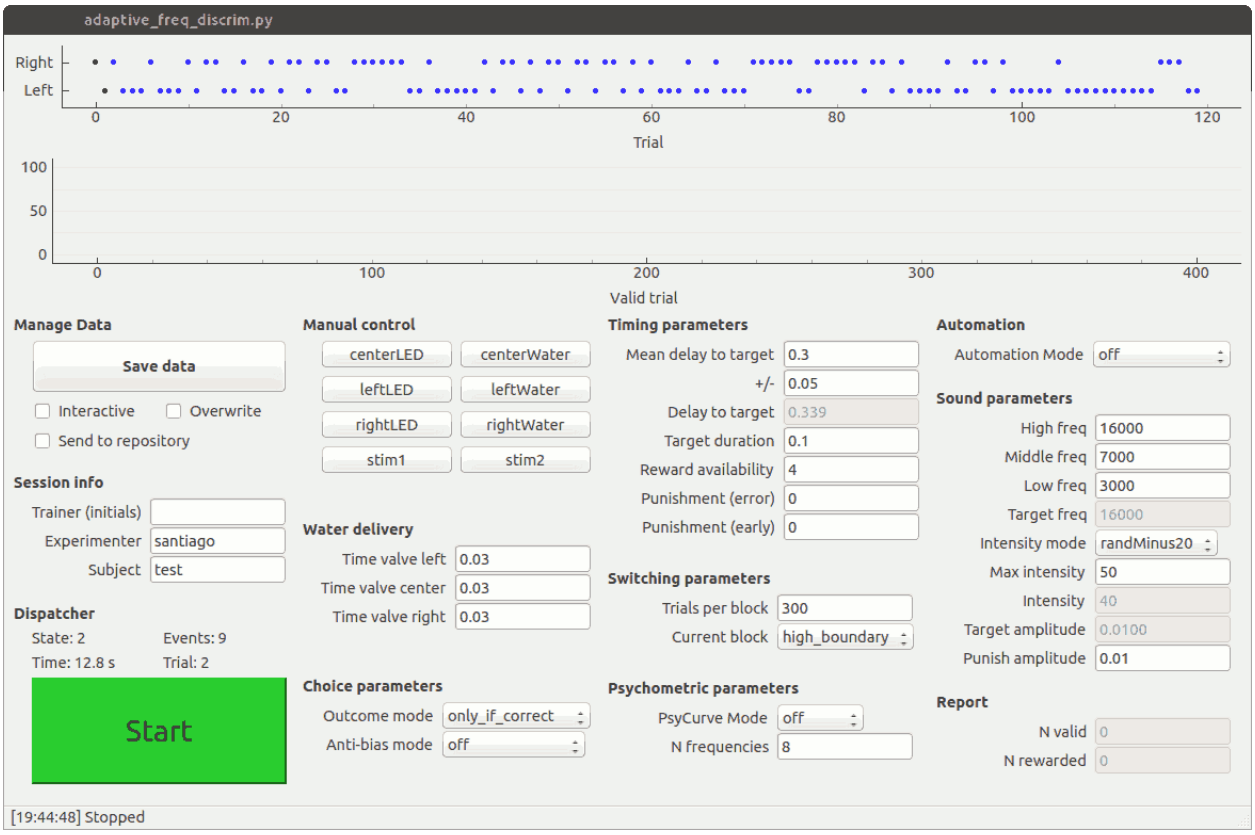
Contents

TASKontrol is an open source framework for developing behavioral experiments for neuroscience research.

It consists of modules written in Python and Qt (via QtPy) for designing behavioral paradigms and providing a graphical user interface to control the experiments. It also includes software that runs in an Arduino Due to provide an interface for detecting external events and triggering stimuli.

TASKontrol was originally developed by Santiago Jaramillo and it is maintained by the Jaramillo Lab at the University of Oregon. The source code can be found in GitHub.

Below is an example of a graphical user interface created with TASKontrol.



NOTE: This documentation is a work in progress and pages for some modules are not yet available. However, these documents should be sufficient to get you started with TASKontrol.

CHAPTER 1

Contents:

1.1 Getting started with TASKontrol

To define an experiment with TASKontrol, you create a *paradigm*. Within a paradigm, you define all the parameters that describe the behavioral task, as well as the graphical user interface to enable starting and stopping the session.

This document explains how to run a simple paradigm using TASKontrol. We will start by running paradigms using the emulator before we try connecting to external interfaces (like an Arduino).

1. First, you need to download and install TASKontrol.
2. Make sure that in your `settings/rigsettings.py` file, you have set `STATE_MACHINE_TYPE = 'emulator'`.
3. Open your favorite editor and save the following Python code into a file (any folder would work). Let's assume you called the file `testparadigm.py`.

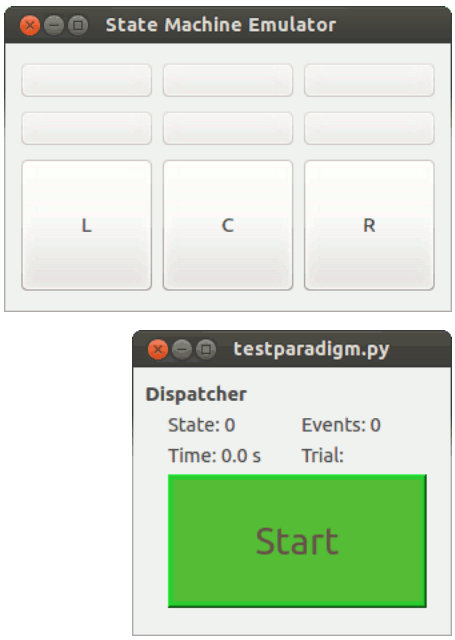
```
from taskontrol.plugins import templates

class Paradigm(templates.ParadigmMinimal):
    def __init__(self, parent=None):
        super().__init__(parent)

if __name__ == "__main__":
    (app, paradigm) = templates.paramgui.create_app(Paradigm)
```

4. In a terminal, go to the folder where you saved the file, and open the paradigm by running the command: `python testparadigm.py`

1.1.1 Interacting with the paradigm



- The command above should open two windows: one with a big "**Start**" button, and one emulator window with multiple buttons.
- When you press the "**Start**" button, the paradigm will run and you will see the time counter increase.
- Pressing buttons in the emulator window will make the "Events" counter in the paradigm window increase, but nothing else should happen.

1.1.2 What is the code doing?

- **Line 1** imports a module that contains paradigm templates. This module will in turn import all necessary modules from Qt (for the graphical interface) and taskontrol (rigsettings, dispatcher, etc).
- **Line 3** is where we define the class for our paradigm, which we call `Paradigm`. In this example, our class is a subclass of the simplest template called `ParadigmMinimal`. To see what is being inherited, look at plugins/templates.py.
- **Line 4-5** are part of the constructor of the class. These lines should appear in any paradigm we create based on a template.
- **Line 7** is a standard Python way of checking if the file is run directly (as opposed to being imported as a module by another file).
- **Line 8** will call the `create_app()` method, which will return:
  - An instance of the `QtGui.QApplication` class (the main class for running Qt applications).
  - An instance of our `Paradigm` class (which gives us access to everything inside our paradigm).

1.2 Defining state transitions

Let's look at how we can detect events and produce outputs. To do this, we need to define the concept of *state transitions*. A key component of every TASKontrol paradigm is the finite-state machine (FSM) which controls how

outputs (stimuli) change given internal or external events. Here is a summary of what this means:

- The system is in only one state at a time.
- When an event occurs, the system transitions from one state to another.
- Events can be triggered externally (*e.g.*, by a button press) or internally (*e.g.*, by a timer).
- Changes in outputs occur when the system enters a state.

1.2.1 Adding state transitions and outputs to the paradigm

The following code shows how we can add states, transitions and outputs to our paradigm. To do this, we override the method `prepare_next_trial()` inherited from our template class. Inside this method, which will be called at the each of each trial, we define states (with corresponding outputs) and transitions for each event.

```
1  from taskontrol.plugins import templates
2
3  class Paradigm(templates.ParadigmMinimal):
4      def __init__(self, parent=None):
5          super().__init__(parent)
6          # The parent class defines self.sm and self.dispatcher used below.
7
8      def prepare_next_trial(self, nextTrial):
9          # -- Set state matrix --
10         self.sm.add_state(name='wait_for_event', statetimer=100,
11                           transitions={'Cin':'light_on'})
12         self.sm.add_state(name='light_on', statetimer=2.0,
13                           transitions={'Cin':'light_off', 'Tup':'light_off'},
14                           outputsOn=['centerLED'])
15         self.sm.add_state(name='light_off', statetimer=0,
16                           transitions={'Tup':'ready_next_trial'},
17                           outputsOff=['centerLED'])
18         self.dispatcher.set_state_matrix(self.sm)
19         # -- Tell the state machine that we are ready to start --
20         self.dispatcher.ready_to_start_trial()
21
22 if __name__ == "__main__":
23     (app, paradigm) = templates.paramgui.create_app(Paradigm)
```

When you run this code and start the session (by pressing the *Start* button), you should see the time counter increase. At this point, pressing the center button (*C*) in the emulator should turn the center light on. You can turn the light off either by pressing the center button again or by waiting 2 seconds (according to the timer defined in line 12).

In this example, an empty state transition matrix `self.sm` is created when executing the constructor of the parent class `templates.ParadigmMinimal` (line 5). One special state is created at this point (`ready_next_trial`), which should be the final state reached at the end of each trial.

The name of each state (*e.g.*, `wait_for_event` in line 10) can be any string defined by the user. The names of events (*e.g.*, `Cin` in line 11) are globally defined for each rig, as described in the next session.

Note: In our implementation of the state machine, outputs have *memory* across states. That is, if an output has been turned on in one state, it will remain on until the system reaches a state that sets this output off.

1.3 Inputs and outputs

The possible inputs (to generate events) and possible outputs (such as lights) for each system are defined when creating a state transition matrix. In general, the list of inputs and outputs for a rig does not change, so we defined them globally in the `settings/rigsettings.py` file.

Below we show how these inputs/outputs can be defined manually or taken from the settings.

Inputs are defined by a Python dictionary such as `{'C':0, 'L':1, 'R':2}` (for center, left and right detectors), which results in two possible events for each input: `Cin`, `Cout`, `Lin`, `Lout`, `Rin`, `Rout`. In this case `Cin` is an event triggered by activating detector C, and `Cout` is generated when the detector is deactivated. An additional event (`Tup`) is always created. This event is triggered when the timer corresponding to the current state ends.

Outputs are defined by a similar Python dictionary such as `{'centerValve':0, 'centerLED':1}`.

When creating an instance of `StateMatrix`, we can also specify the name of the "readystate". When the system reaches this state, it gives the control back from the state machine to the user interface until `dispatcher.ready_to_start_trial()` is called.

Here is an example of how this is done in code:

```
from taskontrol import statematrix
from taskontrol.plugins import templates

class Paradigm(templates.ParadigmMinimal):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.sm = statematrix.StateMatrix(inputs={'C':0, 'L':1, 'R':2},
                                          outputs={'centerValve':0, 'centerLED':1},
                                          readystate='ready_next_trial')
        # The parent class defines self.dispatcher used below.

    def prepare_next_trial(self, nextTrial):
        # -- Set state matrix --
        self.sm.add_state(name='wait_for_event', statetimer=100,
                          transitions={'Cin':'light_on'})
        self.sm.add_state(name='light_on', statetimer=2.0,
                          transitions={'Cin':'light_off', 'Tup':'light_off'},
                          outputsOn=['centerLED'])
        self.sm.add_state(name='light_off', statetimer=0,
                          transitions={'Tup':'ready_next_trial'},
                          outputsOff=['centerLED'])
        self.dispatcher.set_state_matrix(self.sm)
        # -- Tell the state machine that we are ready to start --
        self.dispatcher.ready_to_start_trial()

if __name__ == "__main__":
    (app, paradigm) = templates.paramgui.create_app(Paradigm)
```

Alternatively, we can import rigsettings.py and use the inputs and outputs defined there:

```
from taskontrol import statematrix
from taskontrol import rigsettings
from taskontrol.plugins import templates

class Paradigm(templates.ParadigmMinimal):
    def __init__(self, parent=None):
        super().__init__(parent)
```

(continues on next page)

(continued from previous page)

```
        self.sm = statematrix.StateMatrix(inputs=rigsettings.INPUTS,
                                          outputs=rigsettings.OUTPUTS,
                                          readystate='ready_next_trial')
    # ...lines 10-27 from code above.
```

You can check rigsettings_template.py, to see what inputs and outputs are defined by default.

Note: In this example, we import a couple of additional modules (`statematrix` and `rigsettings`). In previous examples, these were imported by the `templates` module, but here we are using them directly.

1.4 Additional timers (extratimers)

The state machine starts a timer (`statetimer`) when the system enters each state. This timer is used to trigger an event (`Tup`) after a specified period for each state. This is how timed transitions are usually implemented.

Sometimes, it is inconvenient to implement transition logic using only state timers. For example, imagine you need to implement a train of 5 pulses of light. You would need to create an ON state for each pulse, and an OFF state for each pulse:

```
# -- Set state matrix --
self.sm.add_state(name='pulse1_on', statetimer=0.1,
                  transitions={'Tup':'pulse1_off'}, outputsOn=['centerLED'])
self.sm.add_state(name='pulse1_off', statetimer=0.2,
                  transitions={'Tup':'pulse2_on'}, outputsOff=['centerLED'])
self.sm.add_state(name='pulse2_on', statetimer=0.1,
                  transitions={'Tup':'pulse2_off'}, outputsOn=['centerLED'])
self.sm.add_state(name='pulse2_off', statetimer=0.2,
                  transitions={'Tup':'pulse3_on'}, outputsOff=['centerLED'])
...
self.sm.add_state(name='pulse5_on', statetimer=0.1,
                  transitions={'Tup':'pulse5_off'}, outputsOn=['centerLED'])
self.sm.add_state(name='pulse5_off', statetimer=0.2,
                  transitions={'Tup':'ready_next_trial'}, outputsOff=['centerLED'])
```

It would be easier if there was a timer that stays active even if the system has transitioned to another state. This is exactly what *extratimers* do.

Here is an example on how you implement the pulse train using extratimers:

```
from taskontrol import statematrix
from taskontrol import rigsettings
from taskontrol.plugins import templates

class Paradigm(templates.ParadigmMinimal):
    def __init__(self,parent=None):
        super().__init__(parent)
        self.sm = statematrix.StateMatrix(inputs=rigsettings.INPUTS,
                                          outputs=rigsettings.OUTPUTS,
                                          readystate='ready_next_trial',
                                          extratimers=['trainTimer'])
        # The parent class defines self.dispatcher used below.

    def prepare_next_trial(self, nextTrial):
```

(continues on next page)

(continued from previous page)

```
15          # -- Set extra timers --
16          self.sm.set_extratimer('trainTimer', duration=1.5)
17
18          # -- Set state matrix --
19          self.sm.add_state(name='start', statetimer=0,
20                            transitions={'Tup':'pulse_on'}, trigger=['trainTimer'])
21          self.sm.add_state(name='pulse_on', statetimer=0.1,
22                            transitions={'Tup':'pulse_off', 'trainTimer':'end_train'},
23                            outputsOn=['centerLED'])
24          self.sm.add_state(name='pulse_off', statetimer=0.2,
25                            transitions={'Tup':'pulse_on', 'trainTimer':'end_train'},
26                            outputsOff=['centerLED'])
27          self.sm.add_state(name='end_train', statetimer=1,
28                            transitions={'Tup':'ready_next_trial'},
29                            outputsOff=['centerLED'])
30          print(self.sm)
31          self.dispatcher.set_state_matrix(self.sm)
32          # -- Tell the state machine that we are ready to start --
33          self.dispatcher.ready_to_start_trial()
34
35  if __name__ == "__main__":
36      (app, paradigm) = templates.paramgui.create_app(Paradigm)
```

- **Line 8** creates a StateMatrix object with one extratimer called `trainTimer`.
- **Line 16** sets the duration of our extratimer.
- **Line 19** defines the first state, which triggers the extratimer (line 20)
- The system will switch back and forth between states `pulse_on` and `pulse_off` until the `trainTimer` ends making the system transition to the `end_train` state.

1.5 Advanced topics

Here is the documentation for some advanced topics:

1.5.1 State machine implementation

TASKontrol does not control hardware directly. Instead, it provides a client to communicate with a finite state machine running on hardware connected to external signals. The default implementation of the state machine server runs on the Arduino platform, but the framework also provides an emulator to test paradigms. This page describes how the event-driven state machine is implemented.

Arduino programs usually contain two main functions:

- `setup()` which runs once after the board is reset.
- `loop()` which runs over and over from then on.

In our system, `setup()` shuts off all outputs and establishes the connection between the client (the computer running the graphical interface) and the server (the Arduino board). The rest happens in `loop()`. You can see the code in statemachine/statemachine.ino.

- The main function inside `loop()` is `execute_cycle()`, which checks if any inputs have changed or any timers have finished. If any of these events have happened, they get added to an events queue.

- At the end of `execute_cycle()`, we call `update_state_machine()` which will go through the queue transitioning through states.

1.5.2 Low-latency sound

This document explains how we achieve sound delivery with low-latency.

Triggering sounds

The state-machine running in the Arduino Due has the ability to send one output byte through a serial port (`native` port on the board) when reaching a state. See `enter_state()` function in `statemachine.ino`. This serial output serves as a trigger for the computer to generate a sound.

On the computer end, the sound module (object `soundclient.SoundPlayer`) is always waiting for serial inputs. Whenever a serial input arrives, the system will play the sound with index specified by the byte sent.

Note that currently only one sound can be triggered per state.

Sound generation

We use the Python module "pyo" to generate sounds. In addition to having many functions for sound generation, this module works with Jack for low-latency delivery of sounds (see below). However, it is not clear if this module is still maintained, and documentation is limited.

Sound card

To be able to generate sound above 20kHz, we use a Xonar Essence STX sound-card from ASUS.

Achieving low-latency

In our experience, triggers through the serial port are very fast (<1ms). The delay bewteen the Arduino triggering a sound and the sound being produced by the sound-card is due mostly to the computer's sound system.

To achieve lower latencies, we use a combination of:

- Low-latency Linux kernel: package `linux-lowlatency` in Ubuntu.
- Jack: a low-latency sound server. Package `jackd` in Ubuntu.

We first run the Jack server (and keep it open), with the command: `pasuspender -- /usr/bin/jackd -R -dalsa -dhw:STX -r192000 -p512 -n2`

In a paradigm, using the sound-client object (`soundclient.SoundClient`) will start `pyo` and generate all sounds through Jack.

1.6 Reference:

1.7 Classes and methods

- genindex
- modindex

1.8 Help

- reStructuredText Primer.