Control Software for Experimental Setups

A framework to support sophisticated and precise experiments

Christian Henning ✉

This software was developed in the Grewe-Lab to specifically control the execution of fear conditioning (**FC**) and active avoidance (**AA**) experiments. However, the code is general enough to support a much broader range of types of experiments. As long as the core of the setup is a data acquisition solution from NI (NIDAQ), that can be controlled via the corresponding daq-toolbox from Matlab, the software will be able to control stimuli presentations and handle input recordings via digital or analog channels.

Before we start explaining the usage and capabilities, we will lay out some of the basic terminology that is used throughout the documentation and code.

1 Basic Terminology

Experiments are designed in an hierarchical structure, consisting of *cohorts*, *groups*, *sessions* and *subjects*. We will define now these terms, which are used throughout the documentation and give a simple example.

- **Cohorts:** Cohorts are the highest level of distinction of subjects. Each subject may belong to one cohort.
- **Groups:** Groups are the second highest level of distinction of subjects. Each subject may belong to one group. A cohort may be divided into several groups.
- **Sessions:** Sessions are the third level of distinction of subjects. However, sessions typically refer to a temporal evolution of the experiment (e.g., days), such that the same subjects are associated with every session within a cohort/group.
- **Subjects:** A subject is an identifier to distinguish animals within a session, hence, it is the fourth and last level in the hierarchy.
- **Recordings:** A recording refers to the actual process of conducting the experiment with an animal. I.e., a recording has a clearly defined design depending on the cohort, group, session and subject the animal is assigned to. According to this design, the recording acquires the data of the animal.

Note, that the 4-tuple (cohort, group, session, subject) provide a unique identifier for each recording.

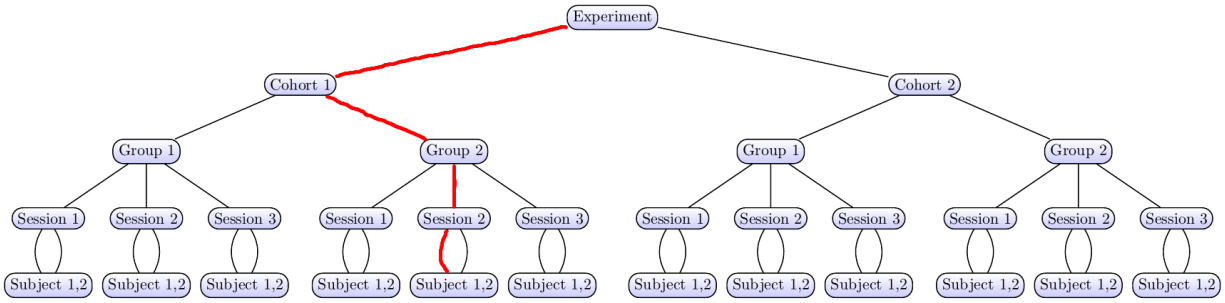

Figure 1: Example experiment design. The multi-edge at the bottom denotes that all subjects within the same cohort, group and session are using the same design.

1.1 Simple Example

Assume an experiment consisting of two cohorts, each containing mice of a different strain. We want to perform simple fear conditioning experiments with two tones A and B. The goal is to assess the differences between these animals.

To make sure that our choice of CS+ does not influence the experiment, we divide each cohort into two groups (for group 1: A is CS+; for group 2: B is CS+).

The experiment will be consisting of 3 sessions on 3 consecutive days. Session 1 is habituation, session 2 is conditioning and session 3 is the testing/readout.

Each cohort contains 4 animals, which we split equally into 2 animals per group. I.e., 8 animals in total are used to conduct the experiment.

Fig. 1 shows a design tree that might depict the experiment. Each path in the tree (for instance the red one) depicts an individual recording. I.e., there are $8 \cdot 3 = 24$ recordings in this experiment.

Note, the numbers in the depicted tree are relative indices. Assume we index a recording r_{ijkl} as follows: i - cohort, j - group, l - session and k - subject. In our experiment, we would use the same subjects within a cohort/group, i.e., for a given i and j, we can state that subject l is the same for all possible sessions $k \in \{1, 2, 3\}$ (we use the same subject in recordings r_{ijkl} $\forall l$ with i, j, k fixed).

2 Design File

The major focus of this document is to explain the structure of a *design file*. The purpose of the design file is to structure the experiment and to define when and how stimuli are presented to the subjects. **Note, that the design has to be generated by the user.** There is no tool yet to support you with the generation of such a file (just a bunch of example design-file-generation scripts)[1].

To be more precise, the design file determines the **passive** design, meaning, that the subject cannot change or influence the presentation of stimuli (as typically the case in FC experiments). If the subject can interact with the setup, we call this an **active** experimental design (for instance, a typical AA experiment will end the stimuli presentation as soon as the subject shuttles). Hence, an experiment is considered to be *active*, if the behavior of the subject can influence the experimental design. For now, we will focus on the *passive* design. The subsection 2.2 will explain how to conduct *active* experiments.

[1] You may also use the auto-generated designs, defined in `control/params.m`. However, they only allow very simplistic designs and are recording-specific (rather than experiment-specific), i.e., each recording will generate a new design file.

The design is stored in a *design folder*. The core of this folder is a `mat`-file (the **design file**), called `experiment.mat`. This file contains a *structure array*, whose format we will explain below. The rest of the design folder may contain design-related files, such as sound-files that are referenced by the design file[2].

Once, the design folder is generated, you can verify its correctness by using the script `misc/experiment_design/check_design_file.m`.

2.1 The format of a design file

We will now give a detailed description of the format and meaning of the structure array, that is stored in the design file `experiment.mat`.

There are two major fields in the design file: `properties` and `design`.

```
{
    "properties":{
        "sound_bit_depth":16,
        "analog_sampling_rate":48000,
        "sound_sampling_rate":48000,
        "experiment_type":"FC"
    },
    "design":{ ... }
}
```

The field `properties` will define several properties that apply for the whole experiment (such as sampling rates for analog events). The property `experiment_type` is just a sanity check, that the type of experiment can be easily accessed. The following values are allowed for this field:

- `FC`: For fear conditioning experiments.
- `AA`: For active avoidance experiments.
- `UN`: For all other types of experiments.

The field `design` contains the actual design tree, as explained in sec. 1. Here is the general structure of the `design` field:

```
...
"design":{
  "cohorts":[
    {
      "names":[ ],
      "infos":{ },
      "groups":[
        {
            "names":[ ],
            "infos":{ },
            "sessions":[
              {
                  "names":[ ],
```

[2]The example design-folder generation scripts in the folder `experiments` often generate in addition to the design file a JSON-file (as they can easily be viewed and verified in a JSON-viewer) and a figure (also for verification). Both files are not needed by the software package.

```
                    "infos":{ },
                    "subjects":[
                      {
                        "names":[ ],
                        "infos":{ },
                        "duration":1320,
                        "shocks":[ ],
                        "sounds":[ ],
                        "events":{ }
                      }
                      ... other subject objects ...
                    ]
                },
                ... other session objects ...
              ]
          },
          ... other group objects ...
        ]
    }
      ... other cohort objects ...
  ]
}
...
```

The `design` field contains a single field `cohorts`, which is a *structure array*. This field (as well as the fields `groups`, `sessions` and `subjects`) contain the special fields `names` and `infos`. The field `infos` appears often inside the design file and is not used by any of the software in this repository. Its meaning is to provide the user with a way to add user-specific information to the design file, that can be utilized by the user its custom code. The field `names` is a list of names. This list has a special meaning, as each entry will define its own cohort-design, which is just a copy of the design defined inside the field `groups`. I.e., the list defines the number of multi-edges (as seen for subjects in figure 1). Instead of defining multiple names inside this list, one could add the same design several times to the `cohorts` struct. But this is just a waste of memory. Hence, whenever two cohorts have the exact same design with respect to the design file (e.g., a cohort of lesioned and a cohort of control animals that should undergo the same experiment), then one can just add several names to the `names` field (implicitly defining multi-edges in the design tree) rather than copying the same design several times. The same accounts for the `names` field in groups, sessions and subjects.

The `subjects` field has additional fields, that are used to define individual recordings. The field `duration` defines the length of a recording in seconds.

The field `shocks` is a remnant from the time when this repository was founded for the purpose of controlling a fear conditioning setup. It simply defines a digital event (for which you could also use the `digital` field within the `events` struct). It is meant to control US events presented to the animal during a recording.

```
...
"shocks":[
  {
    "onset":258.18467787661575,
    "duration":2,
```

```
    "intensity":0.0006,
    "channel":-1,
    "rising":0,
    "falling":2
  },
   ... other shock objects ...
]
...
```

The field `onset` describes the onset of the event since the start of the recording (in seconds). The field `duration` denotes the length of the event (how long is the stimuli presented). The field `intensity` is unused by the software in this repository. In principle, it could be used for programmable foot shockers, but such a feature is not yet implemented. Hence, the field is only reserved for future usage. The field `channel` can be used if shocks should be supplied via various channels. E.g., in an AA experiment, one might have a shock channel for the left and the right shock side. See the `control/params.m` file for more information. By default, this field should be set to `-1`. The fields `rising` and `falling` are necessary for digital events. As we do not specify the digital event as a sequence of zeros and ones (which would also require the specification of a digital sampling rate), we have to specify the timestamps of rising and falling edges. Hence, these two fields are lists in general, where each entry describes the relative time point of a rising resp. falling edge with respect to the start of the event.

For instance, a constant HIGH signal could be expressed as a single rising edge at the beginning of the event and a single falling edge at the end of the event:

```
"rising":0,
"falling":2
```

Similarly, a constant LOW event could be expressed by no rising (and thus no falling) edges:

```
"rising":[],
"falling":[]
```

A more complex event, that is HIGH for 0.5s, then LOW for a 1s and then HIGH again for the rest of the event, could be expressed as follows:

```
"rising":[0, 1.5],
"falling":[0.5, 2]
```

The static methods `toDigitalArray` and `toDigitalEdges` of the class `misc/experiment_design/RecordingDesign.m` can be used to work with such a representation and translate it into a usual digital signal.

The field `sounds` of the `subjects` struct can be used to specify sounds that are played during a recording. It is in principle an analog event as defined by the `analog` field within the `events` struct, but the control software distinguishes between this field and other analog events by allowing to play these kinds of events via the sound card (rather than using the NIDAQ). The advantage of playing sounds via the sound card is, that one doesn't require high sampling rates for the NIDAQ and can therefore save computational resources and memory. On the other hand, a precise timing cannot be assured when using the sound card. A sound event has the following structure:

```
...
"sounds":[
```

```
  {
    "type":"CS-",
    "onset":115.08384913193697,
    "duration":25,
    "infos":{ },
    "data":[ ],
    "filename":"sounds\/cs2.wav"
  },
  ... other sound objects ...
]
...
```

The field `type` is used to distinguish between different sound types later in the evaluation code (for instance, whether the sound is a CS+ or CS- event, when designing an FC experiment). The field `data` can contain the sound signal (which has to adhere the sampling rate as defined in the field `properties`). However, it is recommended to use the field `filename` for that purpose, in which one has to specify a relative path to a sound file containing the sound that should be played in this event.

The field `events` of the `subjects` struct contains arbitrary analog and/or digital events:

```
...
"events":{
  "analog":[
    {
      "infos":{ },
      "description":"Analog event description",
      "events":[
        {
          "infos":{

          },
          "onset":20,
          "duration":10,
          "type":"ANALOG",
          "data": []
        },
        ... other analog event objects from this analog channel
            ...
      ]
    },
    ... other analog event channels ...
  ],
  "digital":[
    {
      "infos":{ },
      "description":"Digital event indicating CS+ Sounds",
      "events":[
        {
          "infos":{

          },
```

```
            "onset":115.08384913193697,
            "duration":25,
            "type":"CS-",
            "rising":0,
            "falling":25
         },
         ... other digital event objects from this digital
            channel ...
      ]
   },
   ... other digital event channels ...
  ]
}
...
```

In this field, one can define multiple analog and digital events. The `analog` field is a list of event channels, each describing a single NIDAQ analog output channel (the same accounts for the field `digital` for NIDAQ digital output channels). Each of these channels is accompanied with a `description` of the channel purpose. Then there is a list of individual event structs in the field `events`. Note, that for analog events, we don't yet support the possibility to use relative filenames to point to external files containing the data.

2.2 How to realize inter-active design?

To the best of our knowledge, there is no way of evaluating the NIDAQ input channels and changing their output design in response to the input in realtime. This attributes to the fact that we work with callbacks, that are scheduled by the operating system (which is in general not realtime). Additionally, we always need to specify the NIDAQ outputs for a certain time window, in which we can't change them anymore. Therefore, we developed an additional device that sits between the NIDAQ and our behavior cage, that can modify the NIDAQ outputs based on the animals behavior. We call this device the *Shuttle Detection Box*, as it was originally developed to detect shuttling behavior and block sounds and shocks based on this behavior.

Manifacturing data for this device can be found in the folder `misc/active_avoidance/shuttle_detection_box`. More documentation about the device and how to build it can be requested from henningc@ethz.ch.

Here, we just list a few example use cases, where this device can be used:

- To control the precise timing of sounds, when using the sound card. Therefore, one specifies sounds that are longer than the event length. Additionally, there exists a precise digital event, that is HIGH when the sound should be played. Without the digital event, the *Shuttle Detection Box* can then block sound events from the sound card, ensuring a precise timing (only works for sound events with no changing temporal structure).

- Shuttle Detection: If the device detects a shuttling behavior during a trial (indicated by another digital signal), it could block the sound and US signals, essentially ending the trial.