3.5 Creation of an *in vivo* data analysis framework

This section describes the result of my work of creating and improving analysis and visualization tools for the large amount of complex data produced during my PhD. This work ultimately lead to the creation of the analysis and visualization framework used during my thesis. This section gives a detailed description of the structure of the software with description of each module and of implementation strategies. Details about the methods used for data analysis of the previously described results are also described here.

3.5.1 The `OCIA` framework

The main tool I used for interacting with and analysing my data was a self-written analysis framework named `OCIA`. This framework will be described in the following sections. This software, written in MATLAB, is currently at version 5.1.10, as it had 5 major iterations where most of the code was re-written or re-organized. The core functions (without the external helper functions) are composed of 573 files and 65'874 lines of code. The entirety of the code is freely available on a public `GitHub` repository, that can be found at the following address:

`https://github.com/blaurenczy/OCIA.git`.

Best practices of programming have been used as much as possible, such as:

1. commenting: 21'094 of the above mentioned lines are commented either in-line or as full comment line, meaning that on average, roughly every 3rd line of code is commented.
2. separating large code into small files following the *MURDER* principle:
   - **M**aintainability: smaller, simpler functions are easier to maintain
   - **U**nderstandability: simpler functions are easier to understand
   - **R**euseability: encourages code reuse by regrouping operations to a separate function
   - **D**ebugability: it is easier to debug simple functions than complex ones
   - **E**xtensibility: code reuse and maintainability lead to functions to extend later
   - **R**egression: reuse and modularization lead to more effective regression testing.
3. modularity of code: there are no hard-coded values, variables or parameters since they would be very hard to find and change; everything that might be changed is a parameter
4. consistent syntax and spacing for an easier human-readability
5. consistent and explicit variable and function naming, as cryptic and/or shortened function and variable names for the sake of brevity can make a code impossible to maintain and understand

Finally, as the first law of programming says: '*Code is more read than written*', therefore sim-

ple solutions and explicit steps of programming have been used along with the mentioned best practices to help future readers of this code to understand it.

Origin and usage

The `OCIA` (pronounced *OKIA*) stands for ***O**nline **C**alcium **I**maging **A**nalysis*. It must be said here that this acronym is purely historical as the framework does not do this any online analysis any more. It originated from the need of an automated data processing pipeline for calcium imaging data. An existing processing pipeline from Dr. Henry Lütcke was originally used and then modified to perform the basic processing steps of calcium imaging analysis. These basic processing steps are described in more details in a later chapter.

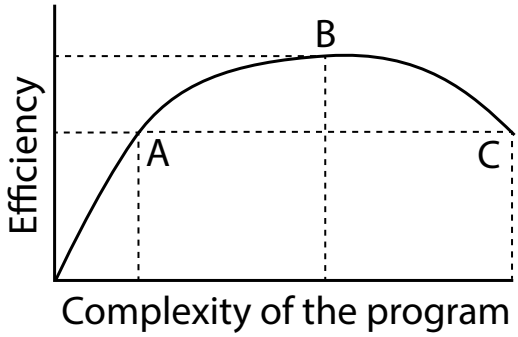

Figure 3.28: *Efficiency versus complexity*

However, it appeared that other parts of the whole analysis pipeline would benefit to be integrated with the automated processing pipeline, for example the non-automated steps of pre-processing and the post-processing analysis steps (see below). The benefit of an integrated analysis pipeline are mainly a gain of time, a better overview of each step, flexibility and increased consistency in the way the data is analysed. Moreover, having a fully automated pipeline leads to the data being processed at speeds allowing an *online* analysis during the experiment, and thus to have a feedback on the quality of the data currently being recorded while doing the experiment. This idea of an online analysis was the beginning of the `OCIA` as a processing tool with a Graphical User Interface (*GUI*) that I used to assess the quality of my recordings during the experiments. This first *quick and dirty* processing tool with a GUI then evolved step by step over four years and many revisions to do much more in a much more sophisticated way and became in its final form the framework described below. A constant worry of this evolution and sophistication, as well as for any task of my PhD, was not to fall on the right side of the *efficiency versus complexity* curve (Figure 3.28).

Efficiency and complexity are related to each other via an inverse U-shaped curve (original idea from Dr. Henry Lütcke). There is an initial fast gain of efficiency (Figure 3.28 A) by creating a program with some complexity, that can automatize sequences of tasks for the user. A more complex program has a gain in efficiency that begins to be smaller (Figure 3.28 B) due to overhead in maintaining and writing code that complies with the other existing features. Finally, creating an extremely complex program might even reduce the efficiency of it (Figure 3.28 C) compared to a simpler software, as staying consistent becomes harder due to the increased complexity and

number of the features. Therefore, one should always consider on which *side* of the curve a program is (or will be), and try to remain '*as complex as needed but not more complex*'.

The OCIA was mainly used by myself to process my own data. However since "*Sharing is caring*", I used the OCIA to process the data sets of several other members of the Helmchen group and some members also used it themselves to process their own data. Altogether and so far, around 10 members of the lab, including myself, used or benefited from at least one module of the framework, most of them by using the calcium imaging related modules.

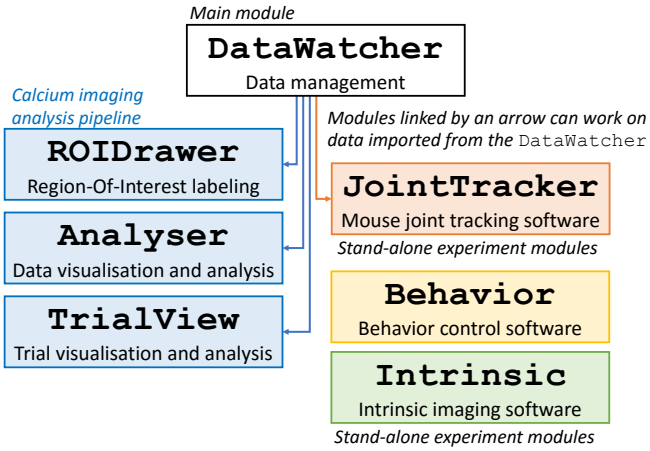

***Figure 3.29: Overview of the OCIA's module**: The DataWatcher is the main module. It handles data management, data saving and loading operations and serves as starting point for most of the other modules (marked by an arrow), including the calcium imaging analysis pipeline. The ROIDrawer, the Analyser and an in a lesser extent the TrialView, are all part of the calcium imaging analysis pipeline. The ROIDrawer is a module providing manual and semi-automatic labelling of neurons for the calcium imaging data analysis. The Analyser provides analysis, plotting and visualization capabilities for any data type using custom analysis functions. The TrialView module is used to analyse mouse behavior trials one-by-one and extract relevant features from behavior movies. The JointTracker is a module integrated with the DataWatcher to perform semi-automatic annotation of mouse joints. The Behavior module is a stand-alone software to control behavior experiments. The Intrinsic module is another stand-alone module, used for the intrinsic imaging experiments.*

3.5.2 An integrated framework with modularity

Given the advantages of the integration mentioned above (speed, flexibility and consistency), I created this framework to merge together these 3 steps of the calcium imaging analysis: the non-automated pre-processing, the automated processing itself and the post-processing analysis. The non-automated pre-processing includes the managing the calcium imaging data sets (see DataWatcher section) and drawing of the Regions-Of-Interest (*ROI*) to define the boundaries of neurons on the images (see ROIDrawer section). The post-processing analysis involves the visualization of the processed data and the actual analysis (sorting, averaging, etc.; see Analyser section).

This whole pipeline had to be implemented in a flexible and modular way, as not all the steps of processing are always required and also not always with the same parameters. Therefore, the pipeline was divided into modules and within each module the parameters to use for each processing steps and analysis were made flexible. Each module is independent but linked to the

others so that the data can be brought from one module to the next.

Since not all modules are always required, the OCIA comes in many different flavours, each with a personalized configuration that includes the required modules. This personalization of the framework is implemented through configuration files that list the requested modules and many other starting parameters, but all of the flavours run on the same core functions of the OCIA. This design allows users to change their configuration file without having to deal with the core functions. Moreover, updates to the core functions or additions of new modules do not disrupt the users, except in the case of major revisions.

3.5.3 The modules and the common features

All modules will be described individually in the following sections. Here is first a brief overview of how they are linked to each other, to complement the module overview (Figure 3.29). The core module, the DataWatcher, is present in all flavors of the OCIA. It serves as starting point for the calcium imaging pipeline but also provides overview of the data sets of any type. The other calcium imaging pipeline modules are the ROIDrawer, the Analyser and an in a less integrated way, the TrialView. The Analyser is also used in other flavors of the OCIA, as it provides plotting and visualization capabilities for any data type and not only calcium imaging. The TrialView module also deals with calcium imaging data but it does not fall in the *classic* calcium imaging analysis pipeline, as its purpose is to give mouse behavior trial visualization and analysis features. The Behavior module is a stand-alone software for the behavior experiments, by controlling and synchronizing the analog and digital inputs and outputs during the experiment; however the Behavior module still uses the common features of the OCIA, detailed further below. The Intrinsic module is similar to the Behavior module, as it is also a hardware controlling stand-alone module, used for the intrinsic imaging experiments. Finally, the JointTracker is also stand-alone modules, also integrated with the DataWatcher, but it performs a specific semi-automatic annotation task described below.

The advantage of having a module-based system is that it saves code, energy and time to implement new modules. The core OCIA has a flexible structure to implement additional modules. This flexible structure has the following features that enable creating new modules easily:

1. **Embedding in the common window**: each module does not need to re-implement the basic features of a window and user interface on its own, they all re-use the common window of the framework (see Figure 3.30).
2. **Easy custom GUI generation**: creating a custom GUI for each module is made easier by a set of common functions to create custom GUI elements (buttons, drop-down menus, check-boxes, tables, etc.). Modules only need to implement the

`OCIA_createWindow_[MODULE_NAME]` function to have a custom GUI created in the main window. Many GUI elements of the `OCIA` framework use the access to the Java properties underlying MATLAB's GUI. These properties allow to better customize the GUI by providing advanced display features or better user interaction capabilities, like mouse drag events.

3. **Mouse and keyboard events**: the common window captures the key press, mouse move, mouse click and mouse drag events, and forwards the actions to each module. Thus, modules only need to implement the actions to do upon the user interactions and not the whole capturing and handling of these interactions (see Figure 3.30).
4. **User feedback**: a common logging, warning and error system is present in all modules, making the handling of errors and warnings easier to code, but also clearer to the user. A *log bar* at the top of the GUI shows messages and warnings to the user. This log bar is common and available in all the modules (see Figure 3.30).
5. **Dynamic function calling**: the framework provides a system to dynamically call custom functions based on their name. This simplifies adding new modules, functions, configuration files, etc. For example, all analysis functions are using the convention `OCIA_analysis_[FUNCTION_NAME].m`. Adding new analysis functions is then easily done by creating a new file named using the same convention. For example a function to do trial averages, `OCIA_analysis_doTrialAvgs.m`, is easy called or referred to by calling the core function `OCIAGetCallCustomFile(this, 'analysis', 'doTrialAvgs')`. This way of referring to functions that have a convention of naming gives a layer of abstraction for the dynamic calling of these functions. In other words, one can then refer to any analysis function within a loop (or *dynamically* call it) without having to hard-code the list of the names of all functions in the code.
6. **Common GUI manipulation**: a set of core function enables to create GUI elements and interact with them in a module-independent way. For example, functions to temporarily disable or enable the whole GUI during data processing are available in any module, like: `OCIAToggleGUIEnableState(this, moduleName, stateOnOrOff)`. Another very useful function can dynamically create a parameter panel based on a customizable configuration (see Figure 3.30). This whole framework contains many configuration files with many parameters and having to write the code for each of the GUI element would be extremely tedious and difficult to maintain, as the configuration files and parameters are often created and modified. Therefore, a single core function, `OCIACreateParamPanelControls` creates a two or three column panel where each parameter from a defined configuration file has its own control GUI element (see Figure 3.30). Since each parameter can be of different type (text, numbers, true/false, lists, etc.) and can be set with different GUI elements (text

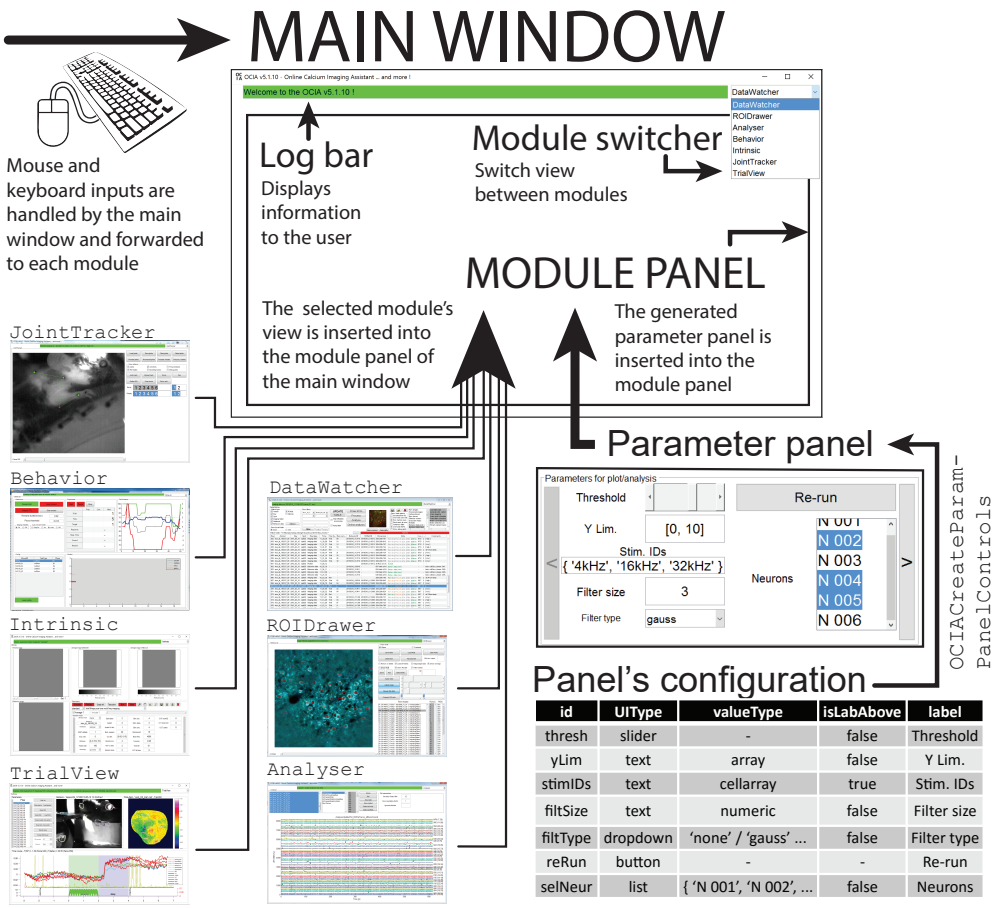

Figure 3.30: Overview of the `OCIA`'s GUI: **Main window:** *the main window contains the log bar, the module switcher and all the modules in a central panel. It also handles the keyboard and mouse events;* **Log bar:** *The log bar displays messages to the user about the computation happening in the software (green background). It can also display warnings (yellow) and errors (red).* **Module switcher:** *A drop-down menu to switch between the different modules. Upon change, the current module becomes invisible and the selected module is displayed.* **Modules:** *Each module only needs to take care of its own specific task, but they all have access to the core functions provided by the framework. All GUI elements have a text tool-tip to explain their function, appearing when hovered with the mouse.* **Parameter panel:** *An example of a core function accessible by all modules. Each module can define a parameter panel configuration, which will then be converted into a parameter panel inserted into the window's GUI, via the* `OCIACreateParamPanelControls` *core function. The configuration for the panel is defined in a dynamic way for each module, using the set of fields shown in the bottom-right table: a unique* id*, the User Interface type* UIType *(slider, text, drop-down, etc.), the input value type* valueType *(text, numeric, array, cell-array, etc.), whether the label should be above or not* isLabAbove*, and the* label *for this element.*

box, slider, buttons, drop-down menu, etc.), having this dynamic creation based on the configuration file rather than hard-coded for each module saves a great amount of code, time and energy. To complement this, the `OCIAUpdateVariablesFromParamPanel` function updates the value of the variable associated with each parameter when the GUI control is used by the user.

3.5.4 Management of complex data with flexibility - `DataWatcher`

As already mentioned above, the amount of data is exploding both in neuroscience and in science in general. This leads to data storage and data transfer problems that better hardware and technology are trying to solve, by providing better data storage capacity and faster transfer rates. However, a very important aspect of this data amount increase is the difficulty to keep an overview of large data sets. Indeed, one of my typical imaging and behavior experiment contains about 9'800 files in about 540 directories or sub-directories. These high numbers of files and directories are simply impossible to overview and remember with a human mind, unless it is assisted by a computer and a good organization software. Such a data management and organization software is referred to as an information system software.

The `OCIA` provides such an information system via the `DataWatcher` module. Although other software for maintaining data organized exist, like `openBIS` (Bauch et al., 2011), I originally wanted a simple and specific system for calcium imaging data. But the `DataWatcher` module, similarly to the whole `OCIA` framework, ended up having a much broader use than originally designed, mainly because of the possibility to customize it. The ability to customize everything in it, led the `DataWatcher` module to be very useful in solving the data management problem, as it can be configured to display and manage any kind of data in a customizable table. The functioning of this module will be described in the sections that follow.

Navigation through hierarchically stored data sets

Data in biology and neuroscience experiments is typically stored in an organized hierarchical manner. Each of the multiple levels of this hierarchy is usually associated with one experimental condition or entity, creating therefore one folder to regrouping all the data related to a single animal, one for a single day, for a recorded region, a recording session, a sample, etc. This hierarchy and organization is most of the time different for each experimenter as very few common standards exist regarding the data storage strategies in neuroscience, or even within single research groups. Therefore, the information system should be made flexible regarding to which levels exist and how they are organized.

As mentioned above, even the data set for a single animal is too big to be viewed all at once (~10'000 files). Therefore, it is required that only a part of the whole data set is displayed at one time. This requires then that only selected parts of the hierarchy tree should be explored and displayed when viewing files. In the `DataWatcher`, users can define through configuration files how their data hierarchy is organized. A list of levels (`watchType` in the code) is defined for each user, where each level can be of two types: either the level is a generic folder, which

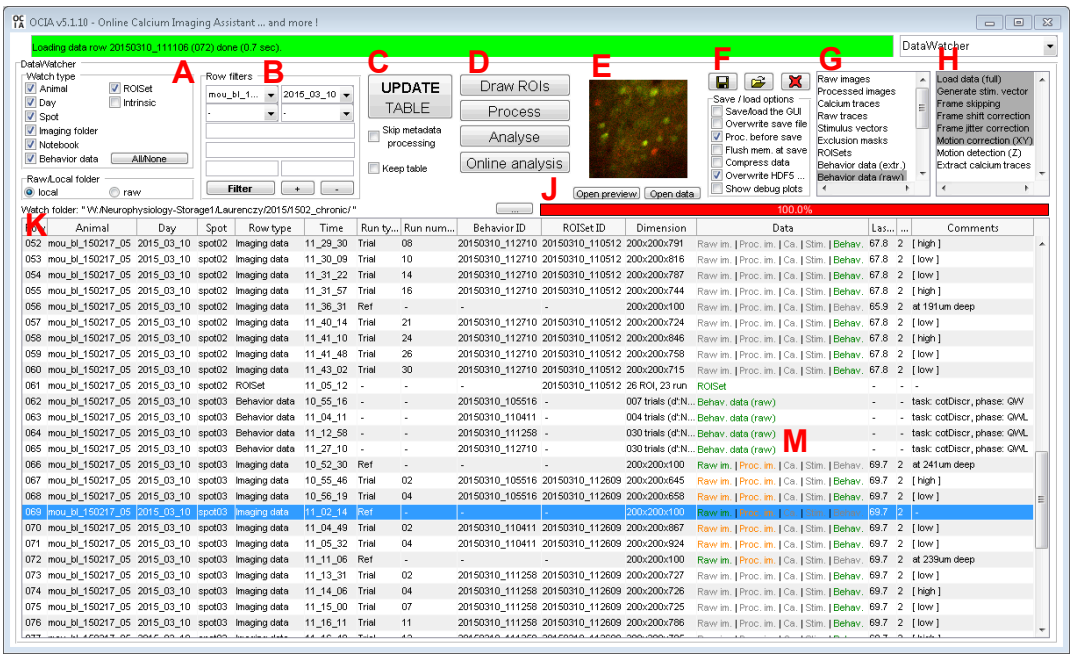

Figure 3.31: ***`DataWatcher` module****: an information system to explore, display and manage any data file of any experiment. Selected data rows can be transferred to the other modules.* ***(A)*** WatchType *filter, used to select which levels of the file hierarchy and which file types should be displayed (see Figure 3.32 for more details).* ***(B)*** Row filter*, enabling the selection of a sub-part of the file hierarchy's tree for display (see Figure 3.32 for more details), but also selection of rows within the table using the* Filter *button.* ***(C)*** *the* Update Table *button triggers the searching in the file system using the* row *and* WatchType *filters and updates the table with the found files and folders.* ***(D)*** *A list of customizable buttons to do any kind of actions with the selected rows, including neuron labelling via the* `ROIDrawer` *(*Draw ROIs*), run an analysis pipeline (*Process*) or display and analyse the processed imaging rows in the* `Analyser` *module (*Analyse*).* ***(E)*** *Image preview of the selected row. Only a few frames of the selected row are loaded for faster preview.* ***(F)*** *Save / load / reset buttons that trigger the saving, loading or clearing of data for the selected rows, with check-boxes to control options for the saving / loading.* ***(G)*** *List of data types available for the displayed rows. Saving, loading and clearing data is only done on the selected data types.* ***(H)*** *List of processing steps for the calcium imaging pipeline triggered by the* Process *button.* ***(J)*** *Loading bar (in red) for the table updating or the processing pipeline, giving feedback to the user about the progress of the requested action.* ***(K)*** *Main table to display the selected sub-set of data files. The table's columns are customizable.* ***(M)*** *The* Data *column has a special coloring system for the loading state of each data type for each row: green text for fully loaded rows, orange for a partially loaded row (e.g. only a couple of imaging frames are loaded for preview), gray for data not loaded.*

just contains another sub-level, or it is a *leaf* level, which contains some actual data files. In the `DataWatcher`'s GUI, users can define which levels they want to display (see `WatchType` filter in Figure 3.31 and in Figure 3.32) and also which specific folder they want to explore (see `Row` filter in Figure 3.31 and in Figure 3.32). The combination of these two filters (levels and specific sub-tree) allows to fully specify the sub-tree that the user wants to explore or display. In addition, the *Keep table* option (Figure 3.31, C bottom) controls whether the previously displayed sub-tree should be cleared from the view or kept when the user requests a search. This allows combining searches of two sub-trees.

To implement this, the `DataWatcher` module goes through such hierarchy using a *recursive*

approach and *regular expressions* (`RegExp`). `RegExp` is a powerful codified language that allows the matching of specific patterns of text. A *recursive* approach implies that the *same* function, `DWProcessFolder`, is called on every level for each encountered item, and this *same* function takes care to recognize which kind of folder each item is, irrespective of the hierarchy level examined. This recursive exploration allows complete flexibility in the way the hierarchy is organized and therefore is suited for any user without the need of hard-coding the specific hierarchy structure for each user. Let us take an example to make the functioning of this recursive function calling explicit.

Usually, the first encountered folder is the *animal* folder. The function examines which kind of folder or file type (called *watchType*) the folder is, based on the folder's name. This association between a folder and a *watchType* is made by testing all *watchType*s sequentially for a possible match against the folder's name, as each *watchType* has an associated regular expression. For example, the *animal* folder *mou_bl_160105_02* would be matched using the *animal watchType*'s regular expression: `^mou_bl_d{6}_d{2}$`. If the folder is a match for one *watchType* and this *watchType* is selected to be displayed via the *watchType* filter, the folder must then pass the *row* filter's test: the folder should also be in the requested sub-tree specified by the *row* filter, in this case the *animal* field of the *row* filter should be set specifically to *mou_bl_160105_02* (our folder's name) or be empty so that any *animal* folder is valid. If the *row* filter is also validating, the folder is further searched. Otherwise if either filter's test is not passed, the folder is not selected for search nor for display and the search moves on.

Once a folder has been validated, the further processing of this folder involves repeating this same examination procedure with all items found in the folder, hence the recursive aspect of this function. Each folder in the sub-tree is therefore searched through, as long as the conditions of the filters are respected. For example, in the *animal* folder mentioned above, let us imagine that there is 2 *day* folders: *2016_03_21* and *2016_04_05*. In addition, let us assume that the *row* filter's *day* field was set to *2016_04_05*. Therefore, when the search process encounters the day *2016_03_21*, it recognizes it as a <*day* folder using the *day watchType*'s regular expression `^d{4}_d{2}_d{2}$`. However, it will not search it further as it is not matching with the *row* filter, set to only accept *2016_04_05*. The other day folder, *2016_04_05*, will on the other hand be recognized as *day* folder and also accepted as being compatible with the sub-tree to search defined by the *row* filter.

Finally, the folders and files found during the search can be *leaf* folders, meaning that they actually contain relevant data files. For example, after the *day* folder mentioned above, there could be a *behavior* folder containing the animal's behavior output files. The search process uses the configuration files to know that this behavior files are to be found in the *behavior* folder. It will

thus search through this *behavior* folder and display in the final table one row for each of these behavior file. The same applies to any *leaf* folder, should it be for imaging files, intrinsic files, etc. These *leaf* files can be read or processed to gain information about them (date, size, type, content, etc.). This information, or *meta-data*, will then be displayed for each item in the overview table, described in the next sub-section. The *Skip meta-data* check-box (Figure 3.31, C bottom) controls whether the meta-data for the found *leaf* data files should be extracted or not. This can be useful as extraction of certain meta-data information (for example file size) can take a great amount of time. The search process can therefore go through the file tree faster by omitting certain steps, depending on the user's need.

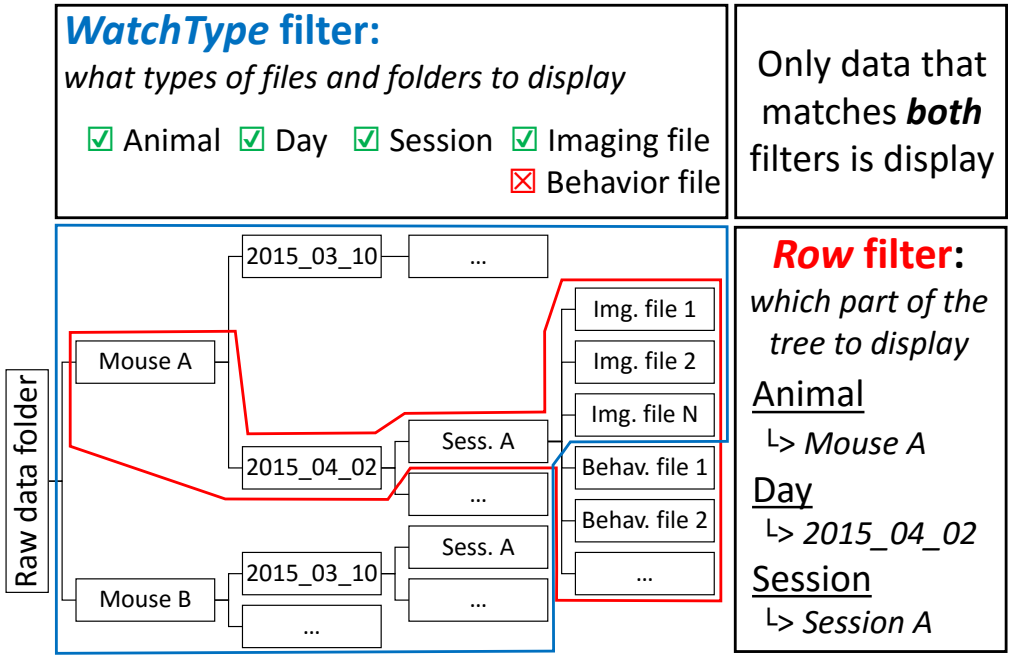

Figure 3.32: The* WatchType *filter *(in blue) is used to select which levels of the file hierarchy and which file types should be displayed. In this example, all levels and files types are requested to be shown, except the behavior files. The* WatchType *filter is combined with the* row *filter (in red), which enables selection of a sub-part of the file hierarchy's tree. In this example, the animal* Mouse A *and the day* 2015_04_02 *and session* Session A *are selected, meaning that only data files from this animal-day-session combination is displayed. For each level (*animal, day, session*), the* row *filter can also be set to empty, meaning that any folder from that level is valid (no filtering on that level). When combining these two filters, only the selected file and folder types for the selected animal/day/session is display (intersection of red and blue area).*

Display of the data set files in a dynamic table

After the search process went through all the files and folders fitting the filters, the gathered information about the file hierarchy's content is then displayed in the main table of the `DataWatcher`. In this table, each row corresponds to a file or folder encountered during the search. For example

in the view presented in Figure 3.31, the first 9 rows correspond to imaging files found in a specific folder (*spot02*) of a specific day (*2015_03_10*) for a specific animal (*mou_bl_150217_05*). Each row is annotated with meta-data during the search and in a post-search annotation step. Information about the origin of the row (animal, day and spot) is added during the search, while other information, like time stamp, file size, row type, etc. are added after the search by a customizable annotation function (`OCIA_annotateTable_*`).

The columns of the table are fully customizable by users, both in terms of their display (ordering, name, etc.) and of their content (what information is stored in each column). This is done through a configuration file that users can easily define before launching the software. The search and annotation steps mentioned before use the user-defined columns to fill the table with information for each row. Since the annotation function is also customizable, the format of the content for each column can also be changed according to the user's own preference. Through this modularity, the table can thus be used for many different purposes, like data management, data processing monitoring or the summarizing of the existing data for an entire project, all without compromising on the precise layout for each user.

The actual data for each row, if any, is not loaded into memory at the search time, it is only referenced in the `DataWatcher`'s main table. The table however contains a field for storing data, but the actual data for each row is only loaded when required. This is implemented in order to avoid loading huge amounts of data into memory at times when this is not wanted. For the same reason, if a row is selected for preview, the data is loaded only partially to memory, just enough for creating a preview. For example in the case of imaging data, only the first few frames of the movie are loaded and then averaged to create the preview. This lowers the memory consumption of the software and increases the speed of preview display, as loading data into memory can be a relatively time-consuming task. Yet, it is still possible and necessary at some times to fully load all the data for a row; this can be done through the `DWLoadRow(this, rowNum, loadType)` function.

The field for storing data mentioned above is somewhat special, as it does not contain only text like all the other field, but it can contain the actual data when it is loaded. The data is stored in this *data* field as a structure containing both the data and information about the type of loading (not loaded, partially loaded or fully loaded). Each row can contain different types of data (raw and processed imaging data, ROI sets, behavior data, calcium traces, etc.) and all of them are stored in parallel in the structure, with their own loading type information. The state of loading for each data type is shown in the display of the table as a colored text (Figure 3.31, M). The text shows whether the data is not loaded (gray), partially loaded (orange) or fully loaded (green). Therefore, users can have an overview of what data is currently loaded in the memory (colored) and what data

is available on the disk but not loaded yet (gray). This feature is an example what can be achieved by accessing the Java components underlying MATLAB, as coloring text in a table requires Java and can not be done with pure MATLAB.

As said above, rows can be selected to be further processed or just to show a preview of their content. For the preview, each row type can implement a `OCIA_getPreview_[ROW_TYPE]` function to display a custom preview, for example a low-resolution image or the average of the frames for a movie or even a color-coded image for the behavior files. This feature mostly uses the partial load of the data as described above. Multiple rows can be selected to be further processed. Selection can be done through two ways: direct selection of the rows by clicking/selecting them or using filters. Filters are a more convenient and faster way to have a specific selection of rows, especially when the table contains already a number of rows.

To create a specific selection, the row filter, also involved in the data set search, can be used. By clicking the *Filter* button at the bottom of the row filter panel (Figure 3.31, B), the displayed rows matching the filtering parameters are selected. Several parameters exist for the row filter, and they are all customizable via a configuration file. In the example view displayed in Figure 3.31, 4 *drop-down*-type filter parameters exist: *animal*, *day*, *spot* (imaging region) and *row type* (imaging row, behavior row, etc.). Each of these have a defined set of values based on what was available but not explored during the search. For example, the search encountered 4 different *animal* folders and 10 *day* folders, but since the filtering parameters were not matching for them, the data files in them was not explored and displayed in the table. However, the search process stored them so that the user can quickly change between different search trees: for example the *day* folders can be quickly changed and a new search launched for another day.

In addition to the *drop-down*-type filter parameters, 4 *text*-type filter exist: general *RegExp*, data load status, row number and run number. The row number and run number parameters can have numbers simple numbers but also a range, for example "'12:15"' or "'[1,2,3:6,7]"'. The general *RegExp* and data load status text filter are more advanced or "'expert-user"' filter parameters as they are using regular expression to match custom columns with custom expressions. In the general *RegExp* field, expressions should use the syntax `COLUMN_NAME = CONTENT AND/OR COLUMN_NAME = CONTENT ...`. A first example not using regular expression but simple text match would be `animal = mou_bl_150217_01 AND rowType = Imaging data AND zoom = 2`, which would find all imaging rows from the specified animal that have a zoom level of 2. The use of regular expression can make this query more precise by adding `... AND comments ~= high`, which would search in the comments text for the "'high"' word, even if it is in the middle of the comments. The data load status text filter uses the same principles, but allows to query the data load type for each row. For example, `data.rawImg.loadStatus = full` would get all the

rows where the raw images are fully loaded. This can be very useful when doing a first processing step and then wanting to move on with only the rows where the processing was successful and the rows are loaded with the output data type. For example, after the calcium imaging processing pipeline, some rows will not pass the quality controls and will therefore not have calcium traces present in their data structure. Specifying `data.caTraces.loadStatus = full` will therefore only select the rows that were fully processed and where calcium data exists for further processing.

Once the wanted rows were selected, several customizable action buttons are available that will perform a custom action only on the selected rows. For example, the calcium imaging processing pipeline can be launched with the *Process* button shown in Figure 3.31 (D). Therefore, the `DataWatcher` can be seen as the starting point of the analysis through these action buttons. The typical workflow of a user would be to open the software, specify some filtering parameters for the file search, do the search, have a look at the existing data files, select some of the data file (manually or using the filters) and then proceed to analyse this selected data.

Finally, a special action button is the *Online analysis* button. Instead of typically moving the selected data files to the next step of processing, this action launches a loop sequence: the table is first updated automatically, then defined rows are selected and processed and some analysis or plotting is performed. The table is then re-updated and another iteration is done, until the online analysis is interrupted. Each steps (search, selection and analysis) can be defined in a configuration file. This mode is very useful when done during experiments, as users can have a fast feedback on what their data actually look like. However this mode requires that all steps can be automatized.

Storage and retrieval of data sets

The `DataWatcher` module takes care of the saving and loading of the data from the disk. This involves the partial and full loading described above, but also the saving and loading of processed data, for example the extracted calcium traces. The file format used by the `OCIA` is the HDF5 format (Hierarchical Data Format version 5). HDF5 has a number of advantages that make it suitable for storing large data sets with complex data in an organized way. HDF5 stores data in a single portable large file that can be compressed and it also enables random reads within the file (all the file does not need to be searched through for finding a specific data set). Furthermore, a number of tools exist to access it from many different programming languages. Furthermore, each data set stored in the file can be annotated with meta-data, so that the origin and experimental context where the data set was created are stored along the data.

The `DataWatcher` stores the data loaded and created in the `OCIA` framework by doing a spare saving and loading. Only the data types requested by the user are saved or loaded. This can be

done using the selection list from the DataWatcher's GUI (Figure 3.31, G). Each data type is saved in its own organized *folder path* within the HDF5 file.

For example, extracted calcium traces for an imaging data file could be stored under the path */animal/day/spot/caTraces/20150310_112400* where as the raw images for the same data file would be under */animal/day/spot/rawImg/20150310_112400*. This allows a nice hierarchical organized storage of files, that can be reused many years later, even more if the appropriate meta-data is saved with the data as mentioned above. The path under which each data set file is saved can be customized by a configuration file. Overall, the DataWatcher creates self-contained data files that are not dependent on the experimenter to be understood and re-used. Furthermore, different HDF5 files can be combined together as the data contained in it is well compartmentalized. It is therefore possible to store the data for all animals in the same single file, that can be compressed for archiving purposes.

3.5.5 Defining Regions Of Interest - ROIDrawer

The ROIDrawer is a module that provides tools for efficiently and quickly label neurons in images from two-photon calcium imaging experiments. It is part of the calcium imaging processing pipeline that the OCIA was developed for in the first place. Originally, this labelling task was done in a much more tedious way using an image processing software called FIJI (Schindelin et al., 2012), part of the ImageJ platform (Schindelin et al., 2015). It quickly appeared that this step was not optimally done by using FIJI and could be made faster, easier and more reliable by integrating it with the rest of the calcium imaging processing pipeline. This however required to create a tool in MATLAB for it, which became the ROIDrawer. The whole calcium imaging processing pipeline will be described in a later section, but the labelling step which is part of it will be described here.

In order to be able to extract relevant calcium signals from the calcium imaging data, a manual or semi-automatic step of neuron labelling is necessary. Indeed, calcium imaging data is recorded from the microscope as images, and these images contain several tens to hundreds of neurons, depending on the imaging field-of-view size. An example image to label is shown in Figure 3.33, (A). Two-photon imaging create an optical section of the neurons present in the brain. Neurons are usually the single units of analysis and therefore the fluorescence collected from them should be grouped together. More technically, each pixel of the image belonging to the same neuron should averaged together in order to obtain a single fluorescence value for each neuron for a given time point. The boundaries of each neuron should thus be delimited or annotated in some form so that an averaging can be performed by the computer in the further steps of the analysis.

The neurons are however not easily defined automatically as their shape is not fully stereotypical, due to their biological diversity in terms of shape and size but also to the optical sectioning

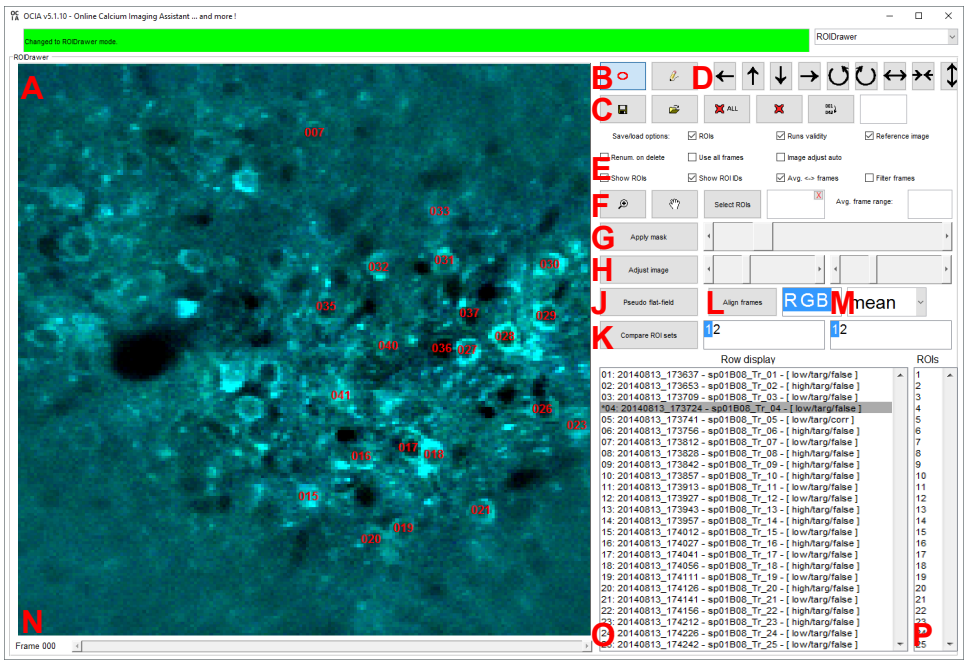

***Figure 3.33:* `ROIDrawer` *module**: a two-photon calcium imaging data Region-Of-Interest defining tool. **(A)** Main display of the image/movie to label. Only half of the ROIs are labelled on this example. **(B)** ROI drawing tools: circular or free-hand. **(C)** Save, load, delete and rename options for the ROIs. **(D)** Transformation tools for the whole ROI set, which helps match ROI sets from the same imaging region on one day and on the next. **(E)** Display option checkboxes. **(F)** Zoom, pan and ROI selection controls. Selection supports range and list input. **(G)** Mask option that shadows all non-labelled pixels. **(H)** Image clipping settings. **(J)** Pseudo-flat-field correction for image smoothing. **(K)** ROI set comparison tool to show the displacement of ROIs from one day to the next. Helps identifying same ROIs for chronic imaging. **(L)** Motion correction on movie frames to sharpen image. **(M)** Color channel control and frame reduction option (mean, max, etc.). **(N)** Current frame number display and frame browser control to run through a movie. **(O)** List of data set rows, each row corresponds to a data file (image or movie). **(P)** ROI selection and display list.*

mentioned above that would give a different picture for a same neuron depending on where it was sectioned. Moreover the labelling provided by the calcium imaging indicator only diffuses to the somata but not to the nucleus of the neurons, giving them a typical *donut*-like shape, a bright ring with a dark center in the middle. This is why a manual labelling (not by a computer) of a Region-Of-Interest (*ROI*) for each neuron is necessary. Some form of automation exist, they will be described below. Furthermore, manual labelling allows greater precision in the inclusion of pixels in the neuron's ROI.

One more problem solved by the `ROIDrawer` is the need to label ROIs for all imaging files. Indeed, calcium imaging experiments typically contain a large number of recordings (movies). In principle, each movie can be slightly different due to movements of the field of view or movements inside the brain tissue. Therefore, labelling done on one movie might not be accurate on a different movie, required the user to label ROIs for each neuron for each imaging file, which can be a tremendous amount of work just for this very fist step of analysis. The `ROIDrawer` and the rest

of the calcium imaging pipeline allows to label ROIs on a single movie and cleverly extrapolate it to relevant imaging data files while still keeping precision in the labelling. This is done by defining a *reference* image on which all neurons are labelled and then *registering* (aligning) all the frames from all the selected imaging movies to the reference image. The registration step is described further below in the calcium imaging processing pipeline section. This way, the labelling process should only be done once per field-of-view, saving a great amount of time for the user.

The `ROIDrawer` provides many tools and features to label neurons in a easy and fast manner, with consistency across different data set files. Here is a list of the features provided by the `ROIDrawer`.

1. Main view: the main view shows the currently labelled image (Figure 3.33, A). The main view supports mouse click events so that users can simply click on the image to label the neurons using the two drawing tools available. The display of the image in the main view can be customized with options like showing or hiding the ROIs or their labels (Figure 3.33, E), changing the shown color channels (Figure 3.33, M), or zooming and panning options (Figure 3.33, F).
2. Drawing tools: elliptic drawing or free hand (Figure 3.33, B). Elliptic drawing provides a quicker labelling as a simple ellipse is shaped around the neuron but it is less precise than the free hand draw which can precisely outline the neuron down to each pixel.
3. Editing: once ROIs have been drawn, it is possible to re-position and re-size them by simply clicking them and then editing them with either the mouse or the keyboard. Furthermore, options to delete and rename ROIs are available to have full control on the ROI set.
4. Saving and loading: labelling of neurons are stored as ROI sets by the `ROIDrawer`. ROI sets contain information about the location and shape of each ROI, their name but also extra information about the *reference* image that was used to defined the ROI set. It also stores information about which imaging file is relevant for this ROI set. Only files marked as relevant for a ROI set are aligned to it. Several ROI sets can be created if the imaging movies for a same field-of-view look very different. This way, all imaging files are always properly labelled. Each of the saving option (ROISet itself, reference image and relevant imaging files) can be activated or not during the saving and loading (Figure 3.33, C), allowing flexible saving and loading operations for the user.
5. Image adjustment: special image adjustment options are available for better seeing the neurons to label. Contrast adjustment options (Figure 3.33, H), *Pseudo-flat-field* correction (Figure 3.33, J) or ROI masking (Figure 3.33, G) are available. The aim of these is to enhance contrast and better show the neurons in regions of the image where a clear outline is difficult to draw visually.

6. Chronic imaging tools: when doing two-photon calcium imaging in a chronic preparation over several days, one needs to identify the same set of neurons across different imaging days. However the shape and size of the field-of-view on one day might look different on the next days. Therefore, a set of ROI set transformation tools is available (Figure 3.33, D) to shift, rotate, expand and shrink the entire ROI set so that it matches the new position of the next day. Furthermore, an ROI set comparison tool is implemented, which shows correspondence between two ROI sets (Figure 3.33, K). This system shows the displacement between ROIs with the same name in ROI sets of the same field of view. This allows easy comparison of ROI sets and guarantees that an ROI corresponds to the same neuron on every imaging day.
7. Shortcuts: each of the function or feature mentioned above can be assigned with a keyboard shortcut to make the tool faster to use.

In summary, the `ROIDrawer` is a quite powerful tool to label ROIs for calcium imaging data, especially in the context of chronic calcium imaging with many imaging files. Having it integrated in the same framework as the calcium processing pipeline allows a very easy transition for users between the different steps of the analysis, as it does not require the usage of an external software, with potential conversion scripts to move from one format to another.

3.5.6 Creation of an automated calcium imaging data processing pipeline

The calcium imaging analysis pipeline was the main and original purpose for which the whole `OCIA` was developed. The purpose of the pipeline is to convert the raw movies recorded by the two-photon microscope into a set of normalized fluorescence time-series for each neuron. This pipeline should ideally be automatized as much as possible, as there are usually a large number of movies and a large number of neurons. The pipeline contains multiple processing steps, which are described further below in this section. Each processing step is not necessarily required for every experiment, and the processing steps can required different parameters depending on the experiment or on the desired output. This is why I originally created the `OCIA` and improved it until it can now offer an automated way of running this calcium imaging processing pipeline, with the possibility to customize which steps should be executed and with which parameters they should be executed.

The processing steps of the calcium imaging data processing pipeline are the following:

1. **Loading of the data**. First, the raw movies should be loaded from the disk. The movies are usually stored in either binary or `TIFF` files. These files should be read and stored in memory. This process is described above in the `DataWatcher` subsection.
2. **Image pre-processing**. Then, the images should be pre-processed to be suitable for the following steps. Indeed, the images can contain multiple types of artefacts due to the record-

ing technique. The removal of these artefacts and the motion correction steps are described separately further below.

3. **Labelling of the neurons**. After the pre-processing of the movies, the neurons should be labelled on a reference image in order to be able to automatically extract the fluorescence time-series. This labelling step is described above in the `ROIDrawer` subsection.
4. **Normalization of the image**. To extract normalized calcium indicator fluorescence time-series, a background normalization step should be done on the movies. Indeed, the fluorescence value recorded in the movies has no unit and is an arbitrary intensity value. Therefore, this intensity should be normalized by subtracting the background intensity from all the images for each movie, in order to have a true "zero" where there is no fluorescence. This is done either by selecting a *background ROI* during the labelling step or by removing a fix value using a percentile of all the intensity values. This latter method is implemented in the `OCIA`, where the first percentile of all pixels from a movie is subtracted from all images.
5. **Extraction of the calcium traces**. Once the movie is normalized, the time-series for each neuron can be extracted using the labelling previously defined. All the pixels within the Region-Of-Interest defined for a neuron are averaged together in each frame. This gives the raw fluorescence F for each time point. This raw fluorescence is then further normalized using the $\Delta F/F$ formula:

   $$\Delta F/F = \frac{F - F_0}{F_0}$$

   The normalization uses F_0 fluorescence level as the baseline fluorescence level. This F_0 can be obtained in various ways from the raw fluorescence F. Possible methods are extracting the mean of the raw fluorescence F, either for the whole trace or for the pre-stimulus time if there are stimulus presentation. A polynomial fit can also be done on the raw trace and used as baseline F_0. The last method, which is the one used in this thesis, is to take a percentile of all the fluorescence values. Typically the 12^{th} percentile for each neuron in each recording is taken as the F_0. This value needs to be however adjusted to the experimental conditions and to the activity of the neurons, as a silent neuron would have its true baseline F_0 at the 50^{th} percentile.

   Note that in the case where ratiometric calcium indicators are used, the ratio R of the two fluorescence channels is calculated and used in place of the raw fluorescence channel F. In this case, the final time-series is usually referred to as $\Delta R/R$.
6. **Generation of stimulus vector**. In the case where stimuli were presented during the calcium imaging, a stimulus vector can be created to mark the stimulus times in relation to the extracted calcium time-series. This stimulus vector can be created by default by the `OCIA` or using custom functions. Each stimulus is encoded in one or several bits of a vector that has

the same length as the calcium time-series. Therefore, each time-point can be marked with an arbitrary number of stimulus type. These stimulus vectors are then used in the `Analyser` module to do analysis and generate plots, for example the peri-stimulus time averages plots.

Automation of the calcium imaging processing pipeline with quality controls

As said above, the execution of all these steps can be controlled by the user, both in terms of their parameters or they can be fully excluded if they are not necessary. This can be done using the GUI controls in the `DataWatcher`'s panel (Figure 3.31, H). Moreover, the inclusion of each processing step in the pipeline and the parameters for each steps can be modified and customized using configuration files. In this way, users can simply select the imaging movies they want to process via the main table of the `DataWatcher`'s panel (Figure 3.31, K) and select the required processing steps (Figure 3.31, H) and launch the fully automated processing pipeline.

The successful completion of the pipeline only requires that the ROIs are already labelled for the selected movies. Otherwise, a warning message will let the user know that this is not done yet. Moreover, the execution of the pipeline can be monitored thanks to a detailed log displayed by the `OCIA` for each processing step. This log also includes quality controls for the steps that require it. Quality controls are an essential requirement in automated processing pipelines, as they show to the user when the data is not suitable for analysis or if something is not conform to the expected outcome. Without quality controls, the pipeline would complete successfully but potentially include erroneous or low-quality data. Parameters for the quality controls, like thresholds of inclusion or exclusion, can be modified using the configuration files.

In addition to the quality controls, debugging plots can be displayed for each processing step. These visual displays of the processing step inputs and outputs help users to better understand what is being done with their data and how each parameter can influence the outcome of the processing pipeline. A feedback on the success of the processing pipeline for each row is finally shown in the *Data* column of the `DataWatcher`'s main table. Rows that do not have a green label in the *Calcium trace* column are imaging movies that had an error in their processing or that did not pass the quality controls. Furthermore, only rows that have valid data can be further used for the analysis in the `Analyser` mode.

The whole pipeline is implemented with a cache-like system that checks what has already been processed and avoids re-processing data that is already existing, saving time and computation power for the user. Finally, the `OCIA` also comes in a *no-GUI* mode, where no graphical interface exists. In addition to this, command line options exist for launching the software with specific tasks or parameters. This allows to run the calcium imaging processing pipeline as batch jobs for

several experiments in parallel, allowing automatized and fast execution of a processing pipeline on multiple computers or on a cluster.

Dealing with image artefacts and motion correction

When recording calcium imaging data with a two-photon microscope in alive animals, many artefacts can be present in the recorded images. Artefacts can be categorized in two classes: recording artefacts and motion artefacts. Both of these artefacts categories can be corrected for, even though sometimes only partially.

The recording artefacts contain artefacts and glitches in the image due to the recording technique or the digitalization of the fluorescence signal into an image. These include for example the first frame of each movie being partially black (no fluorescence) because of the time the shutter needs to move away from the laser's path. Another artefact is the frame jitter, where each line of the image is slightly shifted compared to the next line. This artefact appears if the speed of the X-axis scanning mirror of the microscope does not have a constant speed. The `OCIA`'s calcium imaging processing pipeline provides function to fully remove these artefacts.

The motion artefacts are problems arising due to the recorded biological sample moving under the microscope. This happens in all recording with alive animals and are much more prominent if the animal is awake and even behaving in a task. Motion artefacts are typically corrected in two different ways. First, the lateral motion on the X and Y axis can be corrected on each frame, as described further below. On a second step, the motion on the Z axis, along the optical axis, can only be assessed but not corrected for. Indeed, the motions along the optical axis shift the focus of the microscope up and down, which leads to the recorded imaging plane moving out of focus and therefore not being imaged. Data is therefore missing in the frames where the focus changed too much due to a movement of the animal. Imaging techniques with multiple focal planes being recorded almost simultaneously (*plane hopping*) can compensate for these artefacts by recombining the different planes recorded, but this is not supported by the `OCIA`.

Lateral movement on the X and Y axis can be corrected by aligning the frames to a reference image. Typically, the reference image is defined as the average of a time period in the movie where movement is limited or as the average of all frames (still containing some movements). Each frame of the movie is then aligned to this reference image using well-known registering algorithms. The algorithm used by the `OCIA`, called *TurboReg*, is based on the Java code developed by a group at the `EPFL` (Thevenaz et al., 1998). It is a fast registration method that can be easily parallelized to run faster on computers with multiple cores. Other algorithms exist, some of them allowing to do motion correction on a per-line basis rather than working with the whole frame (Chen et al., 2012).

3.5.7 Analysis and visualization of data sets - `Analyser`

The `Analyser` module takes care of the analysis and visualization of the data. It is designed as an *interactive* analysis and plotting tool. It works mainly with time-series and stimulus vectors, the former being usually extracted from calcium imaging data using the processing pipeline described above. Typically, the images are processing with the pipeline and the extracted calcium time-series are imported into the `Analyser` for analysis and visualization.

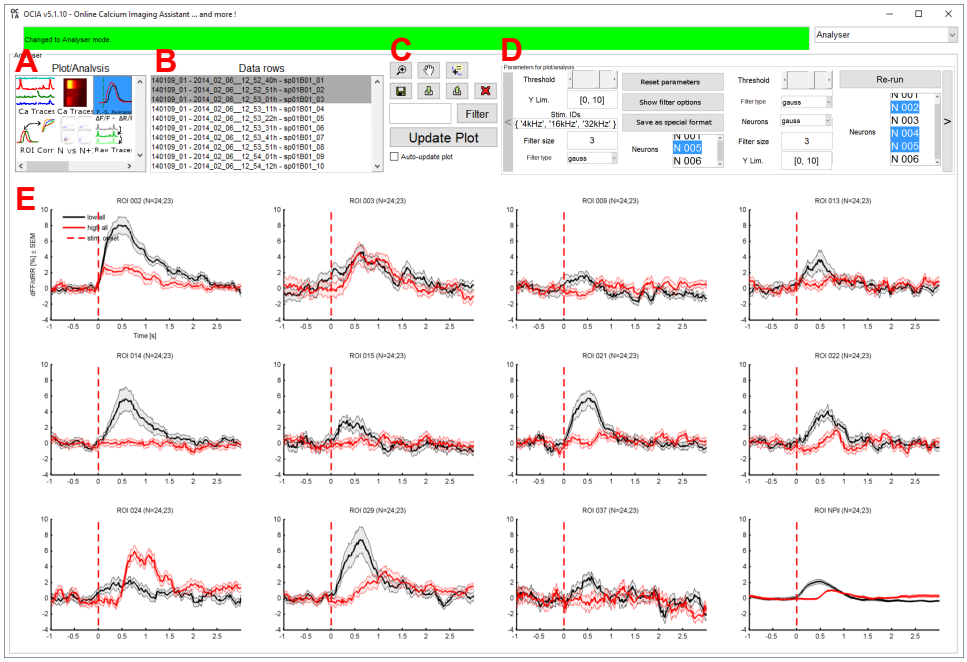

***Figure 3.34: `Analyser` module**: a data visualization and analysis tool. **(A)** Analysis or plot type selection. This list is custom and can be extended simply by adding files to a custom folder, icons being optional. **(B)** List of data set rows, each row corresponds to a data file. **(C)** Top: zoom, pan and data cursor tools to manipulate and examine plots. Middle: save and load options for both the displayed plot but also the analysed output data. Bottom: data set row filter option and update plot button, that re-plots the data or re-runs the analysis. A caching system exists to avoid re-doing analysis with identical parameters. **(D)** Parameter panel for the current plot/analysis. Each plot/analysis has a custom user-defined parameter list that can be easily changed. **(E)** Output display area. The output of each plot/analysis is displayed here into one or multiple plotting regions. The example shown is the peri-stimulus time average response for 11 neurons and the neuropil recorded during a go/no-go discrimination task. Choice of neurons and averaging options can be easily changed using the parameter panel.*

The `Analyser` provides a large set of analysis and plots, which can easily be selected in the GUI (Figure 3.34, A). All these analysis and plots are stored as single function files, creating a base on which to extend. Indeed any custom analysis function can be added to the existing set and will automatically appear in the selection list. In general, each analysis function is also including a plotting function, showing the result of the analysis.

For example, in the view shown in Figure 3.34, a peri-stimulus time average for two different

stimulus types is shown for several neurons. This function builds on the peri-stimulus time averaging function that uses the calcium traces and the stimulus vector. The stimulus type(s) and the neurons to plot can be selected using the parameter panel available in the `Analyser`'s GUI (Figure 3.34, D). Tools are provided to better examine the displayed data: zooming, panning (moving) and data tip (show data value) are available, along with buttons to save the shown plot and/or save the underlying data used to generate the plot. (Figure 3.34, C).

The analysis and exploration of the data is an iterative process, usually done by changing some parameters and running the analysis again. Therefore, an easy way to change the parameters is required. Rather than going in the code and searching for the definition of each parameter, the `Analyser` provides a parameter panel (Figure 3.34, D) to quickly change the parameters without going back to the code.

This parameter panel system is combined with a caching system that automatically saves the results of each analysis when they are executed. This avoids repeating an analysis that has already been done, saving time for the user and increase the interactive aspect of the `Analyser` module. The caching system is aware of the parameter panel and of the dependencies of each analysis to each parameter. Therefore, when an analysis is being run, the caching system checks if a critical parameter has been changed for each analysis step. Only the steps where a critical parameter has been changed will be re-executed, as the change of parameter can lead to a different output of the analysis. In this way, only minimal time is required to examine the effect of an analysis parameter on the output.

3.5.8 Visualization of single trials - `TrialView`

The `TrialView` module is a custom module created for a specific purpose: visualizing single trials for the wide-field calcium imaging with the delayed discrimination experiments. Therefore it is less generic and less customizable as the previously described modules. The precise experimental conditions and resulting data for which this module was created are described in more detail in the *Methods* and *Results* sections. Briefly, animals performed in a auditory *go / no-go* discrimination task with a delayed reporting of the response, combined with wide-field calcium imaging of portion of the cortex.

These experimental conditions, wide-field calcium imaging combined with behavior, produce a great amount of data but more importantly complex data with different types of data sets and different time references that need to be aligned. Indeed, several data types or variables were recorded simultaneously with different means:

1. **Wide-field calcium imaging** recording the cortical activity, which produces a 512-by-512 pixels movie with about 240 frames, usually recorded at 20 Hz, one movie per trial

2. **The behavior monitoring** camera recording the animal, which produces a 720-by-480 pixels movie with about 20'000 frames, usually recorded at 30 Hz, one movie for the entire training session (about 20 minutes).
3. **Licking spout** recording the licking reporting the decision of the animal, which produces a analog time series of about 36'000 data points recorded at a sampling rate of 3 kHz, one time series per trial.
4. **Microphone** recording the played sound stimulation, which produces the same data type as the licking spout (same analog input box).
5. **Wide-field camera trigger signal** recording the triggering time of the wide-field camera, which produces the same data type as the licking spout (same analog input box).
6. **Whisker tracking** camera recording the whiskers' position, which produces high speed images that are converted into a time series down-sampled to the wide-field camera's frame rate, one time series per trial.

A main requirement to analyse this type of experiment was to be able to look at single trials individually with all the above mentioned information and data combined. The aim of the `TrialView` module was to fulfil this requirement by providing a tool that can combine and extract the relevant

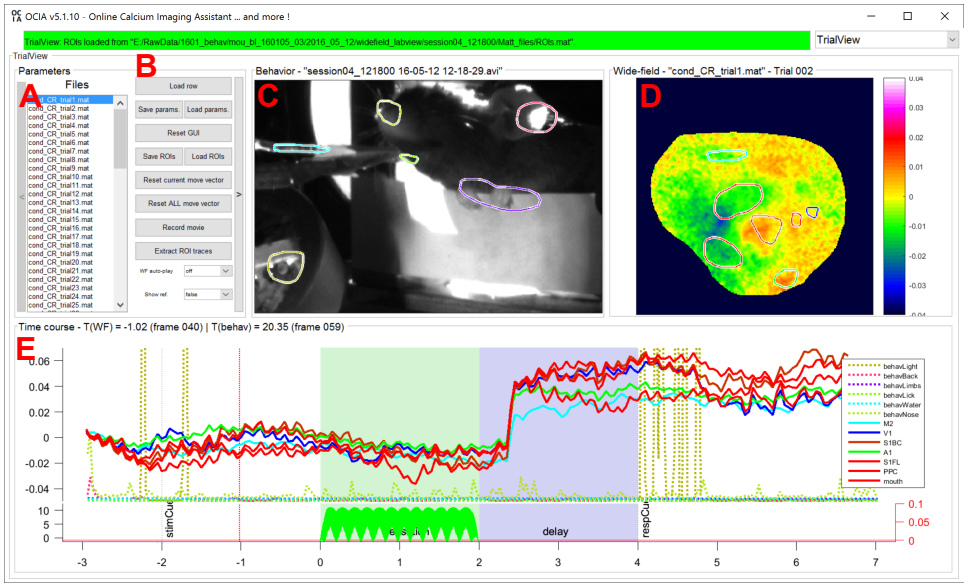

***Figure 3.35: `TrialView` module**: a single trial exploration tool. **(A)** List of the trial files. **(B)** Action buttons: row loading, parameter save and load, GUI reset, ROI set save and load, move vector control options, movie recording, ROI trace extraction. Bottom: display options. **(C)** Behavior movie view with behavior ROIs drawing function. **(D)** Calcium imaging data view with cortical ROI drawing function. **(E)** Time course view of the behavioral variables and the cortical ROI extracted calcium traces. On the bottom, lick trace, sound stimulation, trigger signal and whisker traces can be displayed. On the top part, behavior ROIs and calcium ROIs are overlayed. This allows visual inspection and correlation between the different variables.*

part of each of these data types and display them all in an interactive Graphical User Interface. Given the different time references and frame rates of the different variables, the first challenge accomplished by the `TrialView` module was to bring all these different variables into a common time frame.

When exploring wide-field calcium imaging data, it is necessary to get an idea of both the spatial organisation of activation regions as well as the temporal dynamics of this activation. The `TrialView` module provides this spatio-temporal visualization by showing the spatial organisation in a panel (Figure 3.35, D) and in another panel, the extracted time-series for user-defined Regions-Of-Interest (Figure 3.35, E). This way, users can go back-and-forth between the spatial and the temporal aspect of the recorded cortical activity.

When working with awake behaving animals, each trial is unique as the animals present a large range of variability in their behavior. This variability is often boiled down to a very small amount of variables, in the case of a go / no-go task this can even be just a lick / no lick binary variable describing the whole behavior of the animal. Thanks to the behavior monitoring camera, it was made visible that the animal shows more complexity in the behavior when the body movement is also taken into account and carefully monitored (Dr. Ariel Gilad, personal communication). Moreover, the body movement can greatly influence the activity measured in the cortex with the wide-field imaging. Therefore, looking at single trials is crucial in order to understand the observed cortical activity.

To quantify these body movements, an ROI defining feature was implemented for the behavior movie in a very similar way as for the wide-field imaging movie. Users can define ROIs both for the behavior movie and for the wide-field movie by simply clicking on them and drawing a Region-Of-Interest. In the case of the wide-field movie, the $\Delta F/F$ is computed for the selected region, whereas for the behavior a so-called *frame-to-frame* correlation method is applied. The frame-to-frame correlation $Fr2Fr_{corr}$ for each time point is calculated using the formula:

$$Fr2Fr_{corr}(t_1) = corr(Frame(t_1), Frame(t_2))$$

Frame-to-frame correlation is a very reliable method to detect changes in the behavior movie, therefore being very useful to quantify movements. The two types of time-series are then displayed in a common time frame in the `TrialView`'s GUI (Figure 3.35, E).

Additionally, the `TrialView` module provides a parameter panel, allowing to quickly change parameters for the different display elements (axis limits, colormap, color clipping, synchronization options, etc.) and provides tools to save and load parameters, extracted time-series or an window-capture movie of a whole trial with all the variables displayed in a common window. The time series display panel (Figure 3.35, E) is also made interactive so that users can navigate through the trial

by dragging the mouse or clicking the different time points. Some custom defined action buttons also exist to perform repetitive tasks in an automatized way, like exporting all the behavior vectors for all trials.

Using all these tools, users are able to quickly and efficiently explore trials in an interactive way, enabling the analysis and visualization of the different data types in an integrated framework. This allows comparison and association of the different behavior or experiment variables with the spatio-temporal dynamics of the recorded cortical activity. This module was also used in the analysis of my own data.

3.5.9 Automatization of mouse joint tracking - `JointTracker`

The `JointTracker` module is another custom module created for a specific purpose, similarly to the `TrialView` module. In this case, the aim was to track mouse joints (shoulder, elbow, wrist, etc.) in behavior movies recorded laterally from mice running on a ladder treadmill. The angle between each joint had to be quantified to relate them to the simultaneously recorded motor cortex activity, measured by two-photon imaging. Joints are marked by a black dot to help the tracking.

Previously, a semi-automatic program was used to solve this problem, but the algorithm used by that program was not efficient enough, leading to the manual annotation of the movies, a process that is extremely time consuming. Therefore, a new way of automatically tracking the joints was explored using the *JointTracker* module. The `JointTracker` provides a fully automatic, a semi-automatic and a manual tracking of the mouse joints. It also provides options to pre-processed the videos in order to enhance visibility of the tracked joints.

The automatic and semi-automatic mode use the algorithm described below to track joints. Manual tracking can however sometimes be necessary, in periods of the videos where the joints become less apparent or where the algorithm is not performing optimally. Therefore, a special attention was also made to create a very user-friendly manual tracking experience. Shortcuts and clever mouse interactions were implemented to facilitate the manual tracking and shorten the tracking time. A slider is also present to easily navigate in the movie (Figure 3.36, G), but most of the action are done via the mouse or the keyboard as they are more intuitive and faster input methods.

The algorithm uses a combination of methods to achieve an accurate tracking of the joints. A first reference point needs to be defined for each joint. The algorithm then takes these as reference and tracks the joints on the next frame using always the results of the previous frames, in a sequential manner. The main component of the tracking is the usage of cross-correlations. A reference template image is defined for each joint using the previous frame and a Gaussian

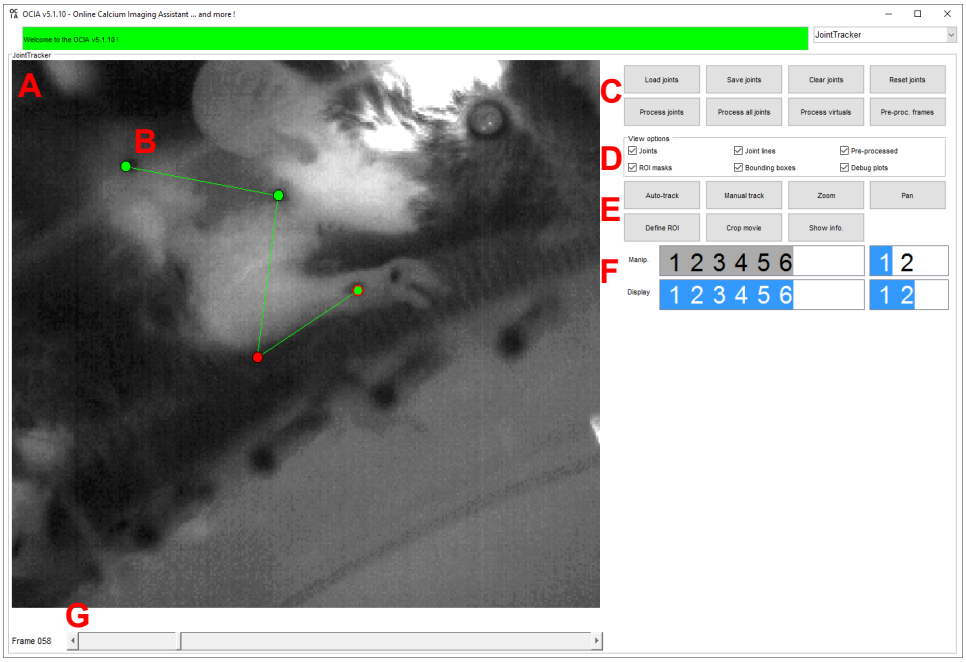

***Figure 3.36: `JointTracker` module**: a mouse joint tracking tool. **(A)** View of the recorded movie. Each joint can be manually clicked and repositioned, or joints can be automatically tracked by the software. **(B)** Joint tracking labels, one point per joint. Different colors label different joint types: manually added joints (black outside, green inside), automatically added joints (red outside, green inside), virtual joints (red inside), etc. **(C)** Save, load and discard options for the joints and processing options for automatically label joints on next frames based on the current joints and user-defined parameters. **(D)** Display options (show/hide joints, joint lines, bounding boxes, debugging plots, etc.). **(E)** Additional view and action buttons: zoom and pan tools, auto tracking, movie pre-processing actions. **(F)** Joint selection options for display and manipulations.*

template matching the joint size. This reference image is then cross-correlated to the current frame, within a delimited region (bounding box) as the joints are not expected to move randomly across the image. The bounding box is dynamically created for each joint, depending on the amount of change in the video since the previous frame. Indeed, a number of change in the video (quantified using the frame-to-frame correlation algorithm described earlier) indicates that the joint might have move more than if the frame is static

The cross-correlation values are weighted using some additional constraints, like the defined distance between the joints which is not supposed to change too much, as the joints are separated by bones that cannot change in length. By combining the cross-correlation values and the expected location of the joints, the most likely current location of the joint is defined and marked as the automatically calculated position of the joint. This position is displayed on the main view (Figure 3.36, A) with dots and lines of different colors, marking the type of joint: automatically annotated, validated, manually tracked, saved joints, etc. (Figure 3.36, B).

This module was not used in the analysis of my own data and was solely created to help other

members of the lab. The main purpose was as said above the mouse joint tracking. However, the flexibility and modularity in the implementation of the algorithm allowed the `JointTracker` to have been used for 2 additional purposes, briefly described below. The flexibility of the algorithm comes among others from the flexibility of its implementation and from the flexible definition of the parameters through a configuration file.

In addition to its original purpose, the `JointTracker` module was used for mouse pup head tracking in a cage and for fiber tip tracking. In the first case, mice were randomly moving in a cage and the amount of movement and the position of the mice across time was tracked using the `JointTracker`, mainly using the manual tracking as the head of the mice was not well defined (black spot on dark background). In the second case, fiber tips of a mouse whisker stimulator were tracked for a jitter quantification experiment. The tips of a glass fiber, marked in black, were semi-automatically tracked using the `JointTracker`, which was possible by slightly adjusting the parameters of the tracking algorithm.

3.5.10 Recording intrinsic signal - `Intrinsic`

The `Intrinsic` module is an experiment controller module created for intrinsic imaging. It uses `MATLAB`'s *Image Acquisition Toolbox* (*imaq*) to grab images from the intrinsic camera. The software is designed to provide feedback to the user during the recording to better control for artefacts and potential problems. It includes a live view panel (Figure 3.37, A), a reference image panel showing the blood vessel pattern (Figure 3.37, B), a set of controls buttons (Figure 3.37, E) to connect/disconnect the hardware and to start/stop the experiment, controls buttons to manipulate and examine the data recorded (Figure 3.37, G, zoom, pan and data tip buttons). Finally, similarly to the other modules described before, an interactive parameter panel also exist in this module to quickly change the experimental parameters. In addition, pre-sets of parameters can also be defined and easily changed to have pre-defined experimental conditions.

Two types of experimental paradigms are implemented: *standard* intrinsic imaging and *Fourier* intrinsic imaging. These paradigms are described in the *Methods* section. Briefly, the standard imaging is trial-based (or repetition-based) stimulus presentation paradigm, where the signal after the stimulus is averaged and normalized to the signal before the stimulus, and the Fourier imaging presents a continuous cycle of a gradual change in a stimulus feature and the signal is then decomposed with a Fourier transform and analysed at the stimulation frequency.

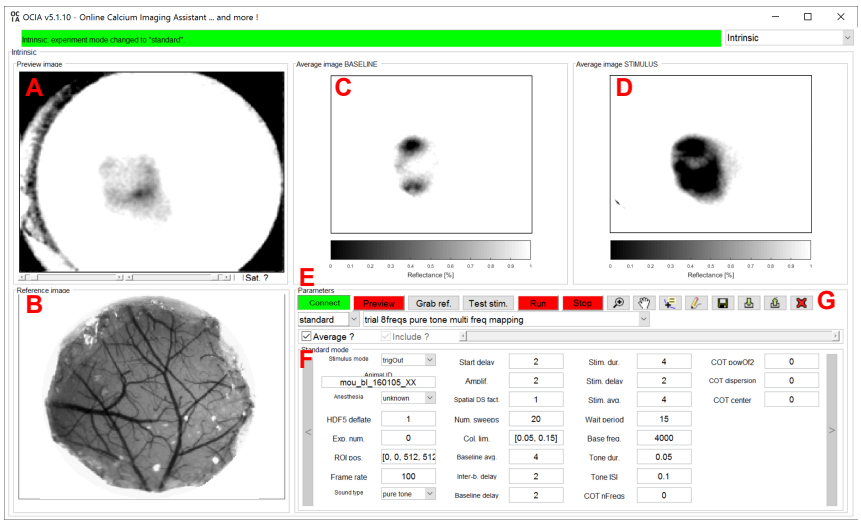

***Figure 3.37:* `Intrinsic` *module**: a stand-alone intrinsic imaging software. **(A)** Live view. **(B)** Reference image view. **(C)** Average image view of the baseline condition (no stimulus). **(D)** Average image view of the stimulus condition. **(E)** Action buttons: hardware connect, preview mode, reference grabbing, stimulation signal test, experiment run and stop. **(G)** View and action buttons: zoom, pan, data cursor and ROI draw tools, save and load buttons for data as images and MAT files. **(F)** Parameter panel controlling acquisition parameters, stimulation protocol and saving options.*

3.5.11 Control of complex behavior training in real time - `Behavior`

The `Behavior` module is another experiment controller module created for the behavior experiments. Many behavior control software are running on the `LabVIEW` environment, which is very suitable for small applications but can become very complex and difficult to maintain for larger software. The `Behavior` module uses `MATLAB`'s *Data Acquisition Toolbox* (*daq*) to read from and write to digital and analog channels. This module was used for all the behavior experiments described in this thesis.

The tasks that are handled by the `Behavior` module are the following:

1. **Communication with hardware**: as said, the `Behavior` modules uses `MATLAB`'s *daq* toolbox to read and write analog and digital data, allowing the software to communicate with the outside world. This is necessary in an experiment controlling software in order to send triggers, send gating controls and collect analog input data
2. **Trial structure and event handling**: the module has an internal representation of the experimental trial and knows what tasks to perform at what time. It has an internal loop that tracks time and processes events as they come. Among others, delay times are waited, stimuli are presented and triggers are sent when appropriate.
3. **Response detection**: the module can detect in real-time whether a response was made by the animal by monitoring an analog input channel, usually the licking port. It can then deliver

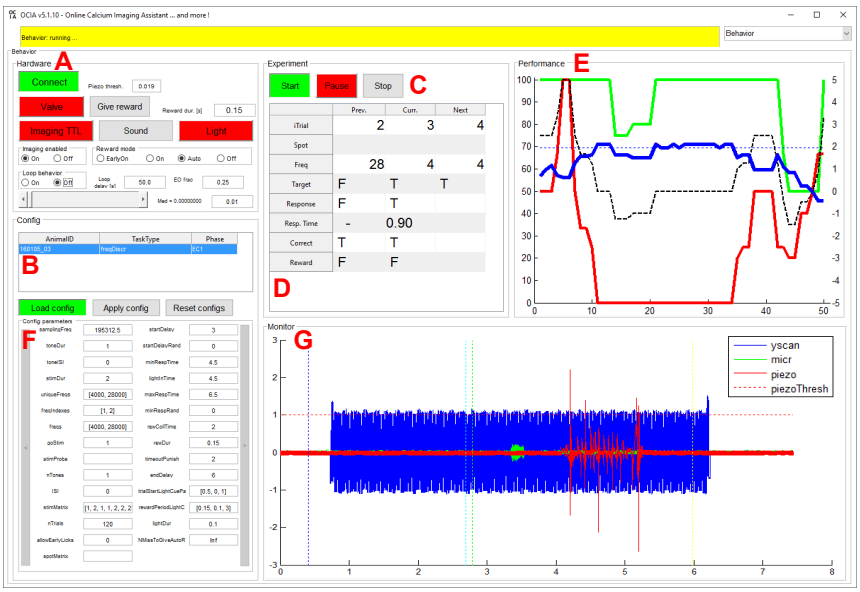

***Figure 3.38:* `Behavior` *module**: a stand-alone behavior control software. **(A)** Hardware panel: action buttons for connecting and setting analog and digital inputs and outputs. **(B)** Configuration panel: a configuration can be selected amongst predefined configurations, usually with one per animal trained. Configurations are stored in a MAT file, which can be generated by a custom script. **(C)** Experiment panel action buttons: start, stop and pause buttons. **(D)** Experiment panel trial table: an overview table of the trial sequence. **(E)** Performance monitoring panel: a plot with running averages of the animal's behavior variables, like hit and false alarm rate, and d' sensitivity index for performance. **(F)** Parameter panel: all parameters controlling the trial sequence and features, like sounds to play, timing, etc., generated automatically from a user-defined configuration file. **(G)** Monitoring panel: real-time display of the recorded analog and digital inputs and outputs, like the licking sensory, the sound, the two-photon's scanner's Y mirror, imaging triggers, etc.*

the appropriate outcome, either a water-reward or eventually a punishment air-puff.

4. **Live display**: in order to monitor the behavior of the animal, a live display of the time-line of the trial and of the experiment is shown (Figure 3.37, C and G). The performance of the animal is also updated after every trial and shown using running average performance curves (Figure 3.37, E).
5. **Live parameter adjustment**: like in any experimental software working with live samples, there is a need of adjusting parameters easily and rapidly during the experiment to adapt to changes in the animal's behavior. This can be done using the parameter panel of the `Behavior` module (Figure 3.37, F).

A key element of such a behavior control experiment is the temporal precision. Indeed, if the rewards or stimuli are not delivered with a sufficient precision, the animal's perception of the task will be perturbed. Therefore, several versions of this module where required to improve and achieve a near-millisecond precision. The main technical challenge was to transition from a procedural sequential approach in the first versions, which had some unexpected delays, to a looping system with *MATLAB*'s `timer` object, which allowed constant time between each event.