General notes on GladOS-Napari

Updates:

* 2024-05-08: initialisation

# (MM)-Controls tab

Controlled via **MMcontrols.py**.

For the tab, a dockWidget\_MMcontrol() is created, which starts a MMConfigUI() class. This has keyword arguments to show certain options (i.e. configugations, stages, ROIs, live-mode, etc).

Generally, these are all ‘semi-hardcoded’ to directly push changes via PycroManager to MicroManager.

Five main code-regions:

1. **Live mode**: has the info to start/stop live mode. In practice, just sets shared\_data.liveMode to True (or False), and just before that, sets core.set\_exposure() to the exposure time in ms
2. **ROI**: set/change the region-of-interest of the camera. Simply has options to reset the ROI (core.clear\_roi(), and zooming the ROI to the center half size (zoomROI, setROI)
3. **Stages**: all movement of XY, and one-dimensional (such as Z) stages. Generally, the layout is created and contains the XYstageLayout() and oneDstageLayout().
   1. **XYstageLayout** takes the current xy device (core.get\_xy\_stage\_device()), and adds arrows in a grid-like pattern to allow the user to change the stage – also has lineEdits to set the value how much each arrow the stage should move. In principle, this finally calls moveXYStage(), which simply calls core.set\_realtive\_xy\_position(deltaX,deltaY), to move the stage relatively to the current position.
   2. **oneDstageLayout** is similar to XY stage devices, except there are multiple QGridLayouts for each one-D-stage found (‘getDevicesOfDeviceType(‘StageDevice’)’), which contains arrows to move the stage and LineEdits to change the value of how much to move. I.e. there can be 5 QGridLayouts in a QStackedWidget, of which 4 are hidden, and they are loaded/hidden based on the dropdown. The stage is moved via moveOneDStage().
4. **MM-configs** shows all the ‘configuration settings’ (groups/presets) which are present in Micromanager. Each group is a row (addRow()), which has a label (addLabel()), and a dropdown, slider, or inputField (inputField should still be implemented). The current values can be updated via updateValueFromMM()/updateConfigsFromMM()
5. **General** has ‘general’ options, i.e. an ‘update all info from MM’, which also does the loading of data, and the ‘store all data’, which calls utils.CustomMainWindow.save\_state\_MMControls().

# Multi-D acquisition

Controlled via MDAGlados.py.

For the tab, a dockWidget\_MDA() is created, which starts a MDAGlados() class, with keywords arguments to instantiate the MDA (i.e. pre-set channels etc).

The MDAGlados has a goal to create a MDA object based on the settings in the GUI. It’s basically a giant wrapper for the pycromanager multi\_d\_acquisition\_events() function. The MDAGlados also pushes the core exposure and shared\_data.mdaModeParams when pressing the ‘acquire’ button.

Four main code-regions:

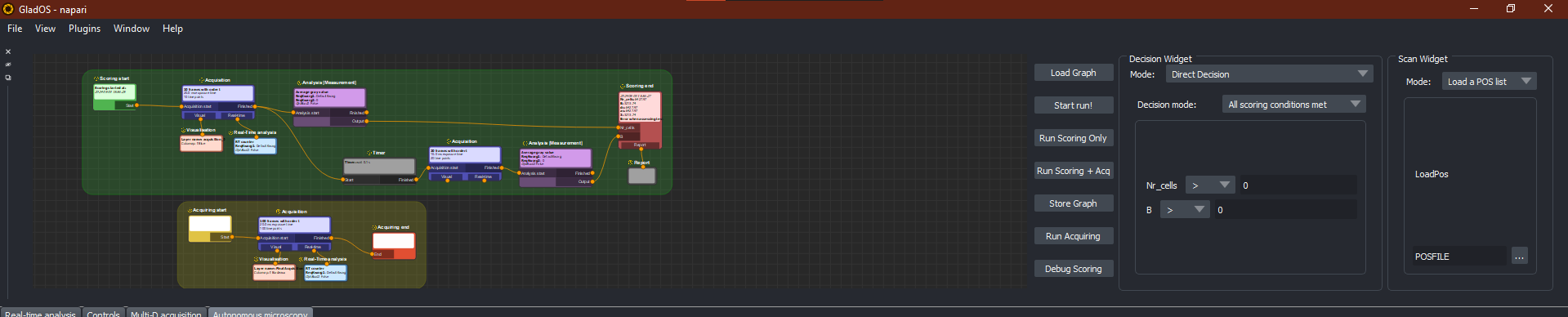
1. **GUI**: settings that the GUI uses to display itself. Most importantly, it allows for only showing part of the full MDA GUI, i.e. only showing the channels window.
2. **Multi-D acquisition logic**: The logci surrounding getting the MDA parameters and pushing these to shared\_data.
3. **Properties**: a somewhat dummy region to help with the GUI grid
4. **List Widgets**: Interactive List widgets for the XY and Channel Lists to allow for nice user-GUIs to add/move/remove positions in a channel or XY list.

# Glados-Nodz-Flowchart

Controlled via FlowChart\_dockWidgets.py.

This controls all Glados-Pycromanager automisation, both the nodz-based UI, as well as the logic behind it.

The goal of this is to create a flowchart plus extra widgets, like this:



The Graph at the left is the GUI-based Nodz environment where the invidivual components are linked together (i.e. at the start, the Scoring is Started, then an Acquisition is started with a certain Visualisation and Real-time analysis, and then this result is going to an Analysis, etc etc).

This is divided in two very distinct parts:

1. Scoring (green block): a custom method is created to score a possible field-of-view.
2. Acquisition (yellow block): if the score is deemed good enough, the acquisition is running.

Here is more information on the graphing area:

* Possible nodes are defined in **defineNodeInfo()** which determines in/out-going connection possibilities and what actions should be run if the node is started.
* After the node is created, the **performPostNodeCreation\_Start()** function is called, which sets the node name, display name, and call actions
* A node is started if all its incoming connections are completed, and the starting node runs its call-action.
* Call-actions are specified in the **Node-specific** region. Example is ‘MMconfigChangeRan’, which changes the configurations of the microscope when the node is started, and then triggers the following nodes when it’s completed.
* A fully completed node should run self.finishedEmits(node) to signal to the attached nodes that it is finished, and the graph is followed like this.

The widgets on the right are two-fold:

1. Scan Widget: determines at which XY(Z) positions the microscope has to be at. At each position, the scoring is ran, and if successfully passed, the acquisition is ran.
2. Decision Widget: determines when a scoring is ‘successfully passed’. Easiest idea is ‘when all scores are above a threshold’, but can be as complex as ‘when at least 10% of the scores is higher than 90% of the randomly sampled area beforehand’ (still to be implemented)

Main code regions of the Nodz-based graphing area:

1. **NodzFlowChart Dialog\_nodz**: Handles creating custom QDialogs when a node is double-clicked. E.g. the ‘timer’ dialog should allow the user to set a certain time. Note that these dialogs are accessed in NodeDoubleClicked() (in the **Node Methods** region).
2. **NodzFlowChart NodzHelperClasses**: Some helper classes for the node graphs. Mostly to create the graphics view, and manage the signals.
3. **NodzFlowChart Node Methods**: The most important node methods. Handles e.g. the creation, initiation, double-clicking, finishing, removing of new nodes. Most important functions:
   1. defineNodeInfo: contains all info for all possible nodes to be created
   2. performPostNodeCreation\_Start: runs when the node is created, and initialises the node fully, and e.g. sets the call action
   3. NodeDoubleClicked: runs when the node is double-clicked, and thus handles the user-defined set parameters.
4. **NodzFlowChart Helpers**: Helper functions for the flowchart, such as changing a node name/color, setting a node name, connecting/removing a plug/socket, etc.
5. **NodzFlowChart Saving Loading**: Handles saving and loading of the Nodz environment. Note that these call saveGraph() and loadGraph\_KM() in nodz\_main.py 🡪 all the important code is in there.
6. **NodzFlowChart Node-specific**: Custom code for specific nodes, i.e. mostly the call-actions of node types are put here. Note that these need to be linked to their call-action in performPostNodeCreation\_Start().
7. **NodzFlowChart runs**: handles partial or full runs of the autonomous microscopy – and thus takes information from the graph, the scoring widget, and the xy scanning widget.
8. **NodzFlowChart GraphArea functions**: functions that are based around the graph area (i.e. context menus and such).

The Scan and Decision widgets are somewhat similar in structure:

They are both created in their class (ScanningWidget/DecisionWidget), and then they create a bunch of advDecisionGridLayout (or advScanGridLayout)s, based on what kind of scanning/decision is ongoing.

All the scanning/decision modes are specified in te init() of the MainWidget, where the ‘\_modes’ has a list of lists with all methods. An advGridLayout is then created with all these modes, where the advGridLayout.mode is set to the mode. The advGridLayout then has to handle all the rest.

Note that the ScanningWidget Modes is ‘1 layer deep’, while the DecisionWidget Modes is ‘2 layers deep’, since it’s directDecision, fullscan, randomscan, stayonfov (i.e. wait at a single spot for an event) options.

The Scanning/DecisionWidgets occasionally run the generic ‘update()’ function of the advGridLayout classes, which need to be linked correctly. They also call, respectively, getPositionInfo() or test\_decision(), which should return the XY(Z) position list or a boolean whether the test/scoring metric is passed at this position.