holypipette Documentation

Release 0.1

holypipette authors

May 11, 2023

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 User manual

1.1.1 Manipulator selection

Select the current manipulator with numbers (1, 2).

1.1.2 Ranges of axes

To measure the range (minimum and maximum) of the axes of the unit and microscope, type M, then move the unit axes and microscope Z to all limit positions (min and max for all axes). This can be done for example by moving to two opposite corners. The same must be done for all units and for the XY stage. For the XY stage, note that the calibration process uses the camera and therefore it must be checked that not only the positions are reachable, but that the image is also acceptable (in particular, there must be light). Type M again to end the measurement process.

1.1.3 Calibration

The coordinate systems of the XY stage and manipulators must be matched to the coordinate system of the camera and microscope. This operation is called *calibration*. The way it works is by matching photos of the pipette tip. Thus, it is very important that the image is clean and sharp at all positions reached by the calibration procedure.

1. Mount a pipette on the manipulator. In principle, any object could be used, the main requirement being that it should be sharp and with a good contrast when seen under the microscope.
2. Put water on the coverslip, as much as possible.
3. Move the microscope as high as possible. If it is an immersion objective, move it up as much as possible while remaining immersed. If it is not an immersion objective, focus on the top of the water drop.
4. Move the pipette manually with the tip in focus, in the center of field.
5. Run the calibration (key: C). The program will first move the XY stage, then move each axis of the unit. If the unit is mounted on the stage, it will also make compensatory movements with the stage when the unit is moved beyond the field of view.

The calibration runs a maximum number of `calibration_moves` exponential moves, which consist in a sequence of movements of the axis, each movement being twice larger than the previous one, with a minimum distance equal to half `stack_depth`. The program also stops if the next movement would not be reachable by the axes, or if it would go beyond the minimum vertical position (Z). The minimum vertical position for the microscope or *floor* can be set with key F. It has not been set, then the program chooses 300 um below the current position. Initially, minimum and maximum ranges of axes are not set, which means that all positions are considered reachable.

Save the calibration with Ctrl+S.

1.1.4 Secondary calibration

When the pipette is changed, or if the pipette appears to be miscalibrated, for example after a large movement, then the pipette should be recalibrated. The idea is that the axis angles should be stable but there can be a shift (translation) in the coordinate systems. To recalibrate, move the pipette in focus and right click on the tip.

1.2 Manipulators

1.2.1 Hardware control

Manipulators are groups of motorized axes, typically an XY stage or an XYZ unit. The basic class is a *ManipulatorUnit*, which depends on a controller. The controller is a set of axes that can be moved independently (for example the Luigs and Neumann controller, which controls up to 9 axes). Example:

```
controller = LuigsNeumann_SM10()
stage = ManipulatorUnit(controller, [7, 8])
```

A *ManipulatorUnit* can be moved with relative or absolute displacements expressed in µm.

1.2.2 Calibrated units

Calibrated units are manipulator units that can be moved in the coordinate system of the camera, called the reference system. A *CalibratedUnit* must be associated to a camera, a microscope Z axis, and can be attached to an XY stage. A *CalibratedStage* is a special kind that a horizontal XY stage (i.e., to be parallel to the focal plane of the microscope). Examples:

```
calibrated_stage = CalibratedStage(stage, microscope=microscope, camera=camera)
XYZ = CalibratedUnit(unit, calibrated_stage, microscope, camera=camera)
```

The position in the camera system is given by $\mathbf{M} \cdot \mathbf{u} + \mathbf{r}_0 + \mathbf{r}_S$, where $\mathbf{u}$ is the position of the manipulator axes, $\mathbf{r}_S$ is the position of the stage in the reference system to which the manipulator is attached, $\mathbf{M}$ is coordinate change matrix and $\mathbf{r}_0$ is an offset.

1.2.3 Movement algorithms

Reference move

The basic move is a *reference_move*. It simply inverts the matrix relation to find the target position in the coordinate system of the manipulator coordinates. However, this is not as simple as it sounds. For some manipulators (including Luigs and Neumann), the resulting displacement is not necessarily a straight line because each axis has a fixed speed independent of the movement. This can result in a broken trajectory; first a diagonal move then a move in the remaining directions. As a result, the pipette could collide with the coverslip or other problems. To avoid this problem, the method has a `safe` option. If True, the method first determines whether the the third axis, which is assumed to be Z, will be moved up or down (see calibration algorithms). If it is going up, then this axis is moved first; otherwise it is moved last. This simple algorithm maximizes the minimum altitude of the trajectory, so as to avoid colliding with the coverslip.

This is only done with absolute moves and not relative moves.

Withdraw

The *withdraw* method moves the first axis to its upper endpoint. This presupposes that the two endpoints have been previously identified.

Focus

The *focus* method moves the microscope Z axis so that the tip is in focus. This does not use an autofocus but rather the calibration system (so the manipulator must be correctly calibrated for this to work).

Safe move

The *safe_move* method moves the manipulator to a target point, with a trajectory that aims at minimizing mechanical interaction with the tissue. It also essentially removes the pipette from the field of view during the approach, which could be helpful if tracking the cell.

If the movement is up, a normal movement is done (with the `safe` option). If it is down, then the trajectory is more complex. First, the manipulator is moved horizontally, then along the first axis of the manipulator. Note that by horizontally, it is meant that the start and end positions are on the same horizontal plane, but the trajectory does not necessarily remain in that plane for reasons explained above; thus the `safe` option is also used.

If the `recalibrate` option is `True` and the movement is at least 500 µm, then the program tries to fix errors in calibration before the target. To this end, the manipulator stops 50 µm before the target, then focus on the tip, automatically recalibrate (see below), then move the focus back, and finish the movement.

Moving a new pipette in the field

This is not fully tested code. The *move_new_pipette_back* method moves a new pipette into the field. This assumes that the calibration is right, except for an offset (due e.g. to the length and slightly different geometry of the pipette). The algorithm is as follows:

1. Move the pipette 2 mm before target position (in the direction of the first axis), which is the center of the microscope view.
2. Take 10 photos at 10 Hz.
3. Calculate the mean standard deviation of the images (more or less the contrast).
4. Move the pipette down by 100 µm along the axis.
5. If the standard deviation of the image differs by at least 20% from the mean calculated previously, stop.
6. Otherwise, go to 4; stop at 5 mm.

In practice, if the pipette is cleaned, this method might not be that useful.

Move and track

The *move_and_track* is used by calibration algorithms. It moves the pipette along one axis, then focus the microscope on the tip using calculation and then template matching. Optionally, it also moves the stage to center the tip. The final image is focused on the tip, but the tip is not necessarily in the center (depending on the precision of calibration). Finally, it returns the position of the tip on screen and focal plane.

Move back

The *move_back* method is used by calibration algorithms. It moves the microscope, manipulator and stage to a given position (previously stored), in a certain order that is intended to avoid collisions. First, the microscope is moved (normally, up), then the manipulator, then the stage. The pipette is then back at the initial position, which is supposed to be in focus in the center of view. Then the pipette is located and refocused, and the pipette position and focal plane are returned.

1.2.4 Calibration algorithms

Calibration consists in determining the matrix $\mathbf{M}$ and the offset $\mathbf{r}_0$, as well as whether the axes go up or down (in Z) in the positive direction.

Recalibration

This assumes that the manipulator is correctly calibrated, except for an offset. The method *recalibrate* updates $\mathbf{r}_0$ assuming that the tip is in the center of view (red cross), or at the given (x,y) position on screen if provided (right-click on the standard interface).

Stage calibration

The stage is assumed to be horizontal, and thus the Z axis of the microscope is not moved. It is assumed that there is an object in focus in the field of view, attached to the stage (pipette, or coverslip). Algorithm:

1. Take a photo of the center of the field: this is the template.
2. Move the first axis by 40 µm, and locate the template in the image: deduce the first column of $\mathbf{M}$.
3. Repeat for the second axis.
4. Using the first estimate of $\mathbf{M}$, move to each of three corners of the image (top left, top right, bottom left), with a safety margin, and locate the template.
5. Calculate $\mathbf{M}$ again based on these three points.

Manipulator calibration

This is the *calibrate* method, plus a number of methods that it calls. The tip must be in focus at the center of view.

Initial steps

1. Calibrate the stage to which it is attached.
2. Take photos of the pipette along the Z axis of the microscope, every 1 µm over distance `stack_depth` (positive and negative).

First estimate

1. Move and track the first axis by a distance equal to half the `stack_depth`. As initially the matrix is zero, there is no predictive move of the focus.
2. Repeat for each axis.
3. Calculate the matrix.
4. Go back to the initial position.

This first very crude estimate is used to calculate the vertical direction of the axes.

Up directions

This is done in method *calculate_up_directions*. It takes the matrix and estimates for each axis whether a positive movement makes the pipette go up or down. Then the minimum reachable Z (coverslip) is determined as 300 µm below the current position, unless it has been specified explicitly (floor position).

Calibration

Each axis is calibrated in turn. For each axis:

1. Double the movement amplitude.
2. Check whether the movement is reachable (which presupposes that ranges have been set).
3. Estimate whether the movement will make the pipette move out from the field of view.
4. Move the pipette and track, and move the stage to compensate if the pipette is out of field.
5. Calculate the relevant column of $\mathbf{M}$, based on camera positions before and after the movement.
6. Repeat `calibration_moves` times.
7. Move back to the initial position.
8. Calculate the relevant column of $\mathbf{M}$, based on camera positions before and after the movement.

Thus, only the last movement (which is the largest one) is actually used to calculate the matrix.

Manual calibration

The *manual_calibration* method takes 4 points chosen by the user, and deduce the matrix from them.

Automatic recalibration

1. Locate the pipette over a depth of +-25 µm, using templates and movements of microscope Z.
2. Update the offset $\mathbf{r_0}$ (recalibration).
3. With option `center`, move the stage and focus so that the pipette tip is centered.

1.3 Pressure control

Holy Pipette can control a pressure controller. Classes inherit the *PressureController* class, which implements three methods. Currently only Elveflow's `OB1` controller is implemented.

Example:

```
controller = OB1()
controller.set_pressure(25, port = 0)
pressure = controller.measure(port = 0)
controller.ramp(amplitude = -100., duration = 1., port = 0)
```

Pressure is mBar.

1.3.1 Fake pressure controller

For development purposes, a *FakePressureController* is implemented. It behaves as a pressure controller, except it is not connected to an actual device.

1.4 Amplifiers

We currently only deal with the *Multiclamp 700B*.

1.5 Automatic patch clamp

These are automatic patch clamp algorithms adapted from the literature.

1.5.1 Main patch clamp algorithm

Amplifier start-up (on Multiclamp 700B):

1. Voltage-clamp.
2. Disable resistance metering and pulses.
3. Compensate pipette (slow and fast).
4. Set pulse amplitude and frequency (default 1e-2 and 1e-2, units unclear).
5. Set zap duration at 1 ms.
6. Do pipette offset (V=0).
7. Set holding potential V = 0.
8. Enable resistance metering (triggers voltage pulses).

Resistance check: 1. Set pressure at `pressure_near` (>0). 2. Do pipette offset (V=0) and wait for 4 s. 3. Check that the resistance is within specified bounds.

Approach:

1. Move the manipulator with a safe move to a distance `cell_distance` above the target position, if specified.
2. Do pipette offset and wait for 2 s.
3. Check that resistance has not increased by 1+`cell_R_increase`.
4. Move down by 1 µm and wait for 1 s (maximum total movement `max_distance`).
5. Measure R. Unless R has increased by 1+`cell_R_increase`, repeat (7).

Sealing:

1. Release the pressure and wait for 10 s.
2. If R is smaller than `1+cell_R_increase` times `R`: go back to approach (7). Note that pressure is now released.
3. Set pressure at `pressure_sealing` (<0).
4. If `R>gigaseal_R`: success (next stage).
5. Ramp V down to `Vramp_amplitude` (default -70 mV) over duration `Vramp_duration`.
6. Wait for at least `seal_min_time`, and until `R>gigaseal_R` (success) or time is out (`seal_deadline`) (failure).
7. Success or failure: release pressure.

Break-in:

1. If `R<gigaseal_R`: failure (seal lost).
2. Increase max pressure by `pressure_ramp_increment`; fail if greater than `pressure_ramp_max`.
3. If `zap` is True, do an electric zap.
4. Do a pressure ramp up to max pressure, of duration `pressure_ramp_duration`; wait for 1.3 s.
5. If `R<max_cell_R`: success.

Ending (also if stopped in the middle):

1. Stop the amplifier: disable resistance metering and pulses; current-clamp.
2. Set the pressure at `pressure_near` (>0).

1.6 Developer guide

1.6.1 Code structure

The code has been separated into a "backend" part and a "frontend" part.

The backend part controls the hardware and implements the algorithms, e.g. what to do to perform a patch clamp on a cell. This code has been written without using any reference to `Qt` and can therefore be reused in simple scripts or in interactive use for testing. However, the use of certain provided functions makes it possible to tightly integrate the code with the GUI, most importantly by making long-running tasks interruptable. For more information, see *Adding new low-level functionality* below.

The frontend is written based on the `Qt` libraries which not only provides the graphical user interface (GUI) but also tools to run things in separate threads and communicate via signals.

This basic structure is summarized in the figure below, showing the names of a few of the most important classes:

GUI classes (first column)

These classes provide the main window the user interacts with. It also defines how the available commands defined in the interfaces (see below) are exposed, e.g. via a Key press or a mouse click. See *Exposing existing functionality in the GUI* below for more details. GUIs that show the camera image should inherit from *CameraGui* which not only provides the GUI for the basic camera image but also an automatic help window, based on the configured mouse/key-bindings, and a viewer for the log file. GUIs that want to expose control of the micromanipulators should inherit from *ManipulatorGui* (which itself inherits from *CameraGui*). Finally, the *PatchGui* supports semi-automatic patch-clamp recordings and itself inherits from *ManipulatorGui*.

Interface classes (second column)

The interface classes provide the link between the GUI and the actual operations defined in the controllers (see below). They all inherit from the *TaskInterface* class and declare and launch commands. Each command has to be implemented in a method and annotated with either the *@command* or the *@blocking_command* decorator. Non-blocking commands (*@command*) are straightforward, short commands that have a direct effect and should be executed from within the main thread. A typical example would be a change in the exposure time of the camera, or storing the current position of the microscope or the manipulators. Blocking commands (*@blocking_command*) are commands that potentially take a long time, such as moving the manipulators to a certain position, and should not be interfered with. For example, during the movement of the manipulator all other commands to move the manipulator should be ignored. To handle starting and ending (potentially by a user abort) of such tasks correctly, the actual task has to be implemented in one or several methods of a *TaskController* (see below). This method should not be called directly, but instead be called via *TaskInterface.execute* which will take care of handling errors and signalling the completion of the task. For more details on this, see *Adding new low-level functionality* below.

Controller classes (third column)

These classes implement the actual tasks by calling the device classes (see below), e.g. by stating that to patch, the cell has to move down until the resistance changes, then set a negative pressure, etc. These classes should be independent from the GUI (e.g. not rely on any `Qt` classes) so that they can be used without it. However, they should inherit from the *TaskController* class and make use of its logging methods (*debug*, *info*, etc.). By using these methods, messages will not only use the general logging system, but also automatically check back whether the user requested the task to be aborted and handle this situation. For more details, see *Adding new low-level functionality* below.

Device classes (fourth column)

These classes expose the hardware functionality in a generic interface, so that hardware can be exchanged without having to change code in the controller classes (see above). By being built on generic classes, actual hardware can also replaced by "fake" devices useful for development. For example, the generic `Camera` class states that all cameras have a `Camera.snap` function that returns the latest camera image. The `FakeCamera` inherits from this class and provides an implementation that returns an artificially generated camera image, while the `uManagerCamera` provides an implementation that returns an actual microscope image via the MicroManager software.

1.6.2 Adding functionality

In the following, we describe how to add various kinds of functionality to existing or newly written classes.

Post-processing the camera image

In GUIs inherting from *CameraGui* (which uses the *LiveFeedQt* widget to display the camera image), "image edit functions" can be used to post-process the camera image. To add such a function, either call *CameraGui*'s `__init__` function with your post-processing function (or a list of such functions) as an argument to its `image_edit` argument, or append to the list stored in `image_edit_funcs` afterwards. The post-processing function or method should take a single argument, the image as a numpy array, and return the post-processed image. Such post-processing functions should only be used to change the image in a way that needs the actual image information; image-independent information that should simply be displayed on top should use the mechanism described in *Displaying information overlaid on the camera image* below.

As an example, consider the following *autoscale* method that scales the contrast of the image to span the full range. It can be implemented in a class inheriting from *CameraGui* as follows:

```
def autoscale(self, image):
    # This assumes a 2D array, i.e. no colors
    if np.issubdtype(image.dtype, np.integer):
        info = np.iinfo(image.dtype)
        total_min, total_max = info.min, info.max
    else:
        total_min, total_max = 0.0, 1.0
    min_val, max_val = image.min(), image.max()
    range = (max_val - min_val)
    # Avoid overflow issues with integer types
    float_image = np.array(image, dtype=np.float64)
    new_image = (total_max - total_min)*(float_image - min_val)/range + total_min
    return np.array(new_image, dtype=image.dtype)

def __init__(self, camera, ...):
    super(..., self).__init__(camera, image_edit=self.autoscale)
```

Warning: Post-processing functions should not change the size and dtype of the image array, other code might directly asks the camera for the size of the video image and then e.g. scale overlays accordingly.

Displaying information overlaid on the camera image

Similar to the image post-processing functions described above, classes inheriting from *CameraGui* can define "display edit functions" which can directly draw on top of the camera image. In the same way as for the image post-processing functions, such functions can be added by either providing them as a `display_edit` argument to `CameraGui.__init__` or by appending to its `display_edit_funcs` attribute. Note that the former will overwrite *CameraGui*'s default overlay, i.e. the cross at the middle of the screen. The overlay function receives a `QPixmap` object and can paint on it using a `QPainter`. It should probably use some transparency to not cover the camera image completely. As an example, the following, admittedly not very useful, code will add a semi-transparent vertical blue line and write "left" and "right" in its two halves:

```
def show_halves(self, pixmap):
    painter = QtGui.QPainter(pixmap)
    pen = QtGui.QPen(QtGui.QColor(0, 0, 200, 125))  #blue, semi-transparent
    pen.setWidth(4)
    painter.setPen(pen)
    c_x, c_y = pixmap.width() / 2, pixmap.height() / 2
    painter.drawLine(c_x, 0, c_x, pixmap.height())
    painter.drawText(c_x / 2, c_y, 'left')
    painter.drawText(c_x + c_x / 2, c_y, 'right')
    painter.end()

def __init__(self, camera, ...):
    super(..., self).__init__(camera)
    self.display_edit_funcs.append(self.show_halves)
```

Adding information to the status bar

The status bar in all GUIs inheriting from *CameraGui* automatically shows long-running tasks or success/failure messages on its left. In principle, GUI code (i.e. code running in the main thread) could show other temporary messages there by calling the `showMessage` function of *CameraGui*'s `status_bar` attribute. The status bar also shows permanent messages on the bottom right, e.g. which micromanipulator is currently in use. A class can add additional information there by calling *CameraGui.set_status_message* which takes a category and a message as its argument. If this function is called again with a new message for the same category, the previous message will be overwritten. Such an update can be triggered regularly by using a `QTimer()`. For example, the following code will update the currently use zoom factor every second by comparing the size of the displayed image (in pixels) with the size of the camera image (a better solution for this use case would be to have this update triggered by a size change instead of with a regular timer).

```
def __init__(self, camera, ...):
    super(..., self).__init__(camera)

    ...

    self.zoom_timer = QtCore.QTimer()
    self.zoom_timer.timeout.connect(self.set_zoom_status)
    self.zoom_timer.start(1000)

def set_zoom_status(self):
    display_size = self.video.pixmap().width()
    image_size = self.camera.width
    zoom = 1.0*display_size/image_size
    self.set_status_message('Zoom', 'Zoom: {:3.0f}%'.format(zoom*100))
```

Exposing existing functionality in the GUI

If a functionality has been defined in the interface class (see *Interface classes (second column)* above, and *Adding new low-level functionality* below), it can be exposed in the GUI. There are two standard methods which also take care of integrating the function with the automatic help window: `register_key_action` and `register_mouse_action`. By convention, these functions should be called in an overwritten version of `CameraGui.register_commands` (which should normally call the parent implementation). The first two arguments of these are the key (as a `Qt` constant, e.g. `Qt.Key_X`), respectively the mouse button (e.g. `Qt.RightButton`) and the modifier. The modifier can either be a `Qt` constant such as `Qt.ShiftModifier` to only trigger the action if the modifier is pressed, or `None` if the action should be triggered independent of the modifier. The modifier `Qt.NoModifier` should be used if the action should only by triggered if the key or mouse button is pressed without any modifier.

> **Warning:** Do not use `Qt.KeypadModifier`, it will be automatically removed from the key event, in particular to avoid problems on OS X where all number key presses carry this modifier.

The third argument is the action to trigger, this should be a method of a `TaskInterface` annotated with `@command` or `@blocking_command` (see *Interface classes (second column)*). Key actions can take an additional `argument`, this can be used to perform a parametrized action, e.g. a move of a given size. Functions that are triggered by mouse clicks automatically receive the mouse position in the camera image (i.e. rescaled and independent of the window size on screen) as an argument. Finally, the optional `default_doc` argument can be set to `False` to not automatically document the action in the Help window. This can be useful when registering many similar commands (e.g. moves of different directions/sizes); they can be summarized with fewer custom help entries by calling `KeyboardHelpWindow.register_custom_action`.

Adding new low-level functionality

Low-level functionality should be added in a `TaskController` class. Such classes should not use any `Qt`-specific code, i.e. should stay independent of the GUI. However, they should make use of the logging functions such as `debug` and `info`, which will automatically check for user-requested cancellations of a running task. Similarly, a task that needs to wait (e.g. for a manipulator that is still moving), should use the `TaskController.sleep` method instead of Python's standard `sleep`.

After adding such functionality, it should be exposed in the `TaskInterface` by adding a method annotated with `@command` or `@blocking_command` (see *Interface classes (second column)*). Finally, this method can then be linked to a keypress or a mouse click in the GUI (see *Exposing existing functionality in the GUI*).

CHAPTER

TWO

API REFERENCE

2.1 holypipette package

2.1.1 Subpackages

holypipette.controller package

Package defining `TaskController` classes. Objects of these classes are responsible for the high-level logic of controlling the hardware, e.g. dealing with the calibration of a manipulator, or defining the procedure for an automatic patch clamp experiment.

Submodules

holypipette.controller.base module

Module defining the *TaskController* class.

exception holypipette.controller.base.**RequestedAbortException**

Bases: Exception

Exception that should be raised when a function aborts its execution due to `abort_requested`.

class holypipette.controller.base.**TaskController**

Bases: *LoggingObject*

Base class for objects that control the high-level logic to control the hardware, e.g. the calibration of a manipulator or the steps to follow for a patch clamp experiment. Objects will usually be instantiated from more specific subclasses.

The class provides several convenient ways to interact with an asynchronously requested abort of the current task. A long-running task can check explicitly whether an abort has been requested with *abort_if_requested* which will raise a *RequestedAbortException* if the `abort_requested` attribute has been set. This check will also be performed automatically if `debug`, `info`, or `warn` is called (which otherwise simply forward their message to the logging system). Finally, tasks should call *sleep* (instead of `time.sleep`) which will periodically check for an abort request during the sleep time.

abort_if_requested()

Checks for an abort request and interrupts the current task if necessary. Can be explicitly called during long-running tasks, but will also be called automatically by the logging functions `debug`, `info`, `warn`, or the wait function *sleep*. :raises RequestedAbortException: If the `abort_requested` attribute is set

delete_state()

Delete any previously saved state. By default, overwrites the `saved_state` attribute with `None`.

has_saved_state()

Whether this object has a saved state that can be recovered with *recover_state*.

Returns

has_state – Whether this object has a saved state. By default, checks whether the `saved_state` attribute is not `None`.

Return type

bool

recover_state()

Recover the state (e.g. the position of the manipulators) after a failure or abort. Has to be overwritten in subclasses.

save_state()

Save the current state (e.g. the position of the manipulators) for later recovery in the case of a failure or abort. Has to be overwritten in subclasses. Should save the state to the `saved_state` variable or overwrite *has_saved_state* as well.

sleep(*seconds*)

Convenience function that sleeps (as `time.sleep`) but remains sensitive to abort requests

holypipette.controller.base.**check_for_abort**(*obj*, *func*)

Decorator to make a function raise a *RequestedAbortException* if `abort_requested` attribute is set.

holypipette.controller.paramecium_device module

class holypipette.controller.paramecium_device.**ParameciumDeviceController**(*calibrated_unit*, *microscope*, *calibrated_stage*, *camera*, *config*)

Bases: *TaskController*

autocenter()

Finds the center of the device.

electrophysiological_parameters()

Reads from the oscilloscope and returns V0, R and Re

move_pipette_in()

It is assumed that the pipette is at working level.

move_pipette_until_drop()

Moves pipette down until Vm drops

partial_withdraw()

holypipette.controller.paramecium_device.**load_data**(*filename*)

Loads a text data file, with the following conventions: - header gives variable names (separated by spaces) - one column = one variable Returns a dictionary of signals

holypipette.controller.paramecium_droplet module

class holypipette.controller.paramecium_droplet.**ParameciumDropletController**(*calibrated_unit*, *microscope*, *calibrated_stage*, *camera*, *config*)

Bases: *TaskController*

autofocus(*position*)

Autofocus on cell at the clicked position

contact_detection()

Moves the pipette down until it touches water.

Algorithm: move down in steps of 5 um until mean intensity has changed by at least one standard deviation.

Note that the focus is untouched (maybe it should follow the tip?).

microdroplet_making()

holypipette.controller.patch module

class holypipette.controller.patch.**AutoPatcher**(*amplifier*, *pressure*, *calibrated_unit*, *microscope*, *calibrated_stage*, *config*)

Bases: *TaskController*

break_in()

Breaks in. The pipette must be in cell-attached mode

clean_pipette()

contact_detection()

patch(*move_position=None*)

Runs the automatic patch-clamp algorithm, including manipulator movements.

sequential_patching()

exception holypipette.controller.patch.**AutopatchError**(*message='Automatic patching error'*)

Bases: Exception

holypipette.devices package

Subpackages

holypipette.devices.amplifier package

Submodules

holypipette.devices.amplifier.amplifier module

class holypipette.devices.amplifier.amplifier.**Amplifier**

Bases: *TaskController*

Base class for amplifiers.

auto_pipette_offset()

Trigger the feature to automatically zero the membrane current.

close()

Shut down the connection to th eamplifier.

current_clamp()

Switch to current clamp mode

resistance()

Returns resistance

set_holding(*value*)

Set voltage clamp value

Parameters
value (*float*) – Voltage clamp value

set_zap_duration(*duration*)

Set the duration for the *zap*. :Parameters: **duration** (*float*) – Duration of the zap in seconds.

start_patch(*pulse_amplitude=0.01*, *pulse_frequency=0.01*)

Initialize the patch clamp procedure (in bath)

stop_patch()

Stops patch clamp procedure

voltage_clamp()

Switch to voltage clamp mode

zap()

"Zap" the cell to break the membrane

class holypipette.devices.amplifier.amplifier.**FakeAmplifier**

Bases: *Amplifier*

"Fake" amplifier that only notes down changes/commands

auto_pipette_offset()

Trigger the feature to automatically zero the membrane current.

close()

Shut down the connection to th eamplifier.

current_clamp()

Switch to current clamp mode

resistance()

Returns resistance

set_holding(*value*)

Set holding voltage or current

Parameters
value (*float*) – Holding voltage or current

set_zap_duration(*duration*)

Set the duration for the *zap*. :Parameters: **duration** (*float*) – Duration of the zap in seconds.

start_patch(*pulse_amplitude=0.01*, *pulse_frequency=0.01*)

Initialize the patch clamp procedure (in bath)

stop_patch()

Stops patch clamp procedure

voltage_clamp()

Switch to voltage clamp mode

zap()

"Zap" the cell to break the membrane

holypipette.devices.amplifier.axoclamp900A_gui module

holypipette.devices.amplifier.multiclamp module

Ported from Clamper ## configure_board and acquire() are unused here

Basic Interface to the MultiClamp 700A and 700B amplifiers.

Note that the MultiClamp Commander has to be running in order to use the device.

For each of the two channels, we have: * command (I or V) * primary * secondary * scope There is also a scope trigger (in the rear)

Gains: actually these are additional gains

class holypipette.devices.amplifier.multiclamp.**MultiClamp**(**channels*)

Bases: object

Device representing a MultiClamp amplifier with two channels or more.

Parameters

channels – List of MultiClamp channels. If none, a single 2-channel Multiclamp is assumed.

acquire(**inputs*, ***outputs*)

Send commands and acquire signals.

Parameters

- **inputs** – A list of input variables to acquire. From: V1, I1, Ve1, V2, I2, etc (electrode potential)
- **outputs** – A dictionary of commands. From: V1, I1, V2, I2…

configure_board(*theboard*, *primary=None*, *secondary=None*, *command=None*)

Configure an acquisition board.

Parameters

- **primary** – A list of names of connections on the board for the primary signal, for each channel.
- **secondary** – A list of names of connections on the board for the secondary signal, for each channel.
- **command** – A list of names of connections on the board for the command signal, for each channel.

class holypipette.devices.amplifier.multiclamp.**MultiClampChannel**(***kwds*)

Bases: *Amplifier*

Device representing a MultiClamp amplifier channel (i.e., one amplifier with two channels is represented by two devices).

Parameters

kwds – Enough information to uniquely identify the device. If there is a single device, no information is needed. If there is a single amplifier with two channels, only the channel number (e.g. `channel=1`) is needed. If there are multiple amplifiers, they can be identified via their port/device number (700A) or using their serial number (700B).

acquire(**inputs*, ***outputs*)

Send commands and acquire signals.

Parameters

- **inputs** – A list of input variables to acquire. From: V, I, Ve (electrode potential) A maximum of two inputs.
- **outputs** – A dictionary of commands. From: V, I. Only one command!

all_devices = None

auto_bridge_balance()

auto_fast_compensation()

auto_pipette_offset()

Trigger the feature to automatically zero the membrane current.

auto_slow_compensation()

check_error(*fail=False*)

Check the error code of the last command.

Parameters

fail (*bool*) – If `False` (the default), any error will give rise to a warning; if `True`, any error will give rise to an `IOError`.

close()

Shut down the connection to th eamplifier.

configure_board(*theboard*, *primary=None*, *secondary=None*, *command=None*)

Configure an acquisition board.

Parameters

- **primary** – A connection name on the board for the primary signal.
- **secondary** – A connection name on the board for the secondary signal.
- **command** – A connection name on the board for the command signal.

current_clamp()

Switch to current clamp mode

dll_path = 'C:\\Program Files\\Molecular Devices\\MultiClamp 700B Commander\\3rd Party Support\\AxMultiClampMsg'

find_amplifiers()

Return a list of all amplifier devices (each described by a dictionary, see `_identifiy_amplifier`).

Returns
amplifiers – A list of all detected amplifier devices.

Return type
list of dict

get_bridge_resistance()

get_fast_compensation_capacitance()

get_meter_value()

get_primary_signal()

get_primary_signal_gain()

get_pulses_amplitude()

get_pulses_frequency()

get_secondary_signal(*signal*)

get_secondary_signal_gain()

get_slow_compensation_capacitance()

null_current()

resistance()

Returns resistance

resistance_meter_state()

select_amplifier()

Select the current amplifier (will be called automatically when executing command such as `MultiClamp.voltage_clamp`.

selected_device = None

set_bridge_balance(*state*)

set_fast_compensation_capacitance(*capacitance*)

set_holding(*value*)

Set voltage clamp value

Parameters
value (*float*) – Voltage clamp value

set_primary_signal(*signal*)

set_primary_signal_gain(*gain*)

set_primary_signal_hpf(*hpf*)

set_primary_signal_lpf(*lpf*)

set_pulses_amplitude(*amplitude*)

set_pulses_frequency(*frequency*)

set_secondary_signal(*signal*)

set_secondary_signal_gain(*gain*)

set_secondary_signal_lpf(*lpf*)

set_slow_compensation_capacitance(*capacitance*)

set_zap_duration(*duration*)
Set the duration for the *zap*. :Parameters: **duration** (*float*) – Duration of the zap in seconds.

start_patch(*pulse_amplitude=0.01*, *pulse_frequency=0.01*)
Initialize the patch clamp procedure (in bath)

stop_patch()
Stops patch clamp procedure

switch_holding(*enable*)

switch_pulses(*enable*)

switch_resistance_meter(*enable*)

voltage_clamp()
Switch to voltage clamp mode

zap()
"Zap" the cell to break the membrane

holypipette.devices.camera package

Submodules

holypipette.devices.camera.camera module

holypipette.devices.camera.lucam module

holypipette.devices.camera.lucamcamera module

holypipette.devices.camera.opencvcamera module

holypipette.devices.camera.umanagercamera module

holypipette.devices.gamepad package

Submodules

holypipette.devices.gamepad.gamepad module

holypipette.devices.manipulator package

Submodules

holypipette.devices.manipulator.calibratedunit module

A class to handle a manipulator unit with coordinates calibrated to the reference system of a camera. It contains methods to calibrate the unit.

Should messages be issued? Also ranges should be taken into account

Should this be in devices/ ? Maybe in a separate calibration folder

class holypipette.devices.manipulator.calibratedunit.**CalibratedStage**(*unit*, *stage=None*, *microscope=None*, *camera=None*, *config=None*)

Bases: *CalibratedUnit*

A horizontal stage calibrated to a fixed reference coordinate system. The optional stage refers to a platform on which the unit is mounted, which can be None. The stage is assumed to be parallel to the focal plane (no autofocus needed)

Parameters

- **unit** (*ManipulatorUnit for this stage*)
- **stage** (*CalibratedUnit for a stage on which this stage might be mounted*)
- **microscope** (*ManipulatorUnit for the microscope (single axis)*)
- **camera** (a camera, ie, object with a `snap()` method (optional, for visual calibration))

calibrate()

Automatic calibration for a horizontal XY stage

equalize_matrix(*M=None*)

Equalizes the length of columns in a matrix, by default the current transformation matrix

mosaic(*width=None*, *height=None*)

Takes a photo mosaic. Current position corresponds to the top left corner of the collated image. Stops when the unit's position is out of range, unless width and height are specified.

Parameters

- **width** (*total width in pixel (optional)*)
- **height** (*total height in pixel (optional)*)

Return type
A large image of the mosaic.

reference_move(*r*)

Moves the unit to position r in reference camera system, without moving the stage.

Parameters

- **r** (*XYZ position vector in um*)
- **safe** (*if True, moves the Z axis first or last, so as to avoid touching the coverslip*)

reference_relative_move(*r*)

Moves the unit by vector r in reference camera system, without moving the stage.

Parameters
r (*XYZ position vector in um*)

class holypipette.devices.manipulator.calibratedunit.**CalibratedUnit**(*unit*, *stage=None*, *microscope=None*, *camera=None*, *config=None*)

Bases: ManipulatorUnit

analyze_calibration()

Analyzes calibration matrices.

auto_recalibrate(*center=True*)

Recalibrates the unit by shifting the reference frame (r0). The pipette is visually identified using a stack of photos.

Parameters
center (*if True, move stage and focus to center the pipette*)

calculate_up_directions(*M*)

Calculates up directions for all axes and microscope from the matrix.

calibrate(*rig=1*)

Automatic calibration. Starts without moving the stage, then moves the stage (unless it is fixed).

calibrate2()

Automatic calibration. Second algorithm: moves along axes of the reference system.

delete_state()

Delete any previously saved state. By default, overwrites the `saved_state` attribute with `None`.

equalize_matrix(*M=None*)

Normalizes the transformation matrix so that each column corresponds to a 1 um move. By default the current transformation matrix is used. This requires a calibrated stage.

focus()

Move the microscope so as to put the pipette tip in focus

load_configuration(*config*)

Loads configuration from dictionary config. Variables not present in the dictionary are untouched.

locate_pipette(*threshold=None*, *depth=None*, *return_correlation=False*)

Locates the pipette on screen, using photos previously taken.

Parameters

- **threshold** (*correlation threshold*)
- **depth** (*maximum distance in z to search; if None, only uses the depth of the photo stack*)
- **return_correlation** (*if True, returns the best correlation in the template matching*)

Returns
x,y,z

Return type
position on screen relative to center

manual_calibration(*landmarks*)

Calibrates the unit based on 4 landmarks. The stage must be properly calibrated.

move_and_track(*distance*, *axis*, *M*, *move_stage=False*)

Moves along one axis and track the pipette with microscope and optionally the stage.

Parameters

- **distance** (*distance to move*)
- **axis** (*axis number*)

Returns
x,y,z

Return type
pipette position on screen and focal plane

move_back(*z0*, *u0*, *us0=None*)

Moves back up to original position, refocus and locate pipette

Parameters

- **z0** (*microscope position*)
- **u0** (*unit position*)
- **us0** (*stage position*)

Returns
x,y,z

Return type
pipette position on screen and focal plane

move_new_pipette_back()

Moves a new (uncalibrated) pipette back under the microscope

normalize_axis(*column*)

Normalizes a column so that it corresponds to a 1 um move. This requires a calibrated stage.

pixel_per_um(*M=None*)

Returns the objective magnification in pixel per um, calculated for each manipulator axis.

recalibrate(*xy=(0, 0)*)

Recalibrates the unit by shifting the reference frame (r0). It assumes that the pipette is centered on screen.

recover_state()

Recover the state (e.g. the position of the manipulators) after a failure or abort. Has to be overwritten in subclasses.

reference_move(*r*, *safe=False*)

Moves the unit to position r in reference camera system, without moving the stage.

Parameters

- **r** (*XYZ position vector in um*)
- **safe** (*if True, moves the Z axis first or last, so as to avoid touching the coverslip*)

reference_move_not_X(*r*, *safe=False*)

Moves the unit to position r in reference camera system, without moving the stage, but without moving the X axis (so this can be done last).

Parameters

- **r** (*XYZ position vector in um*)

- **safe** (*if True, moves the Z axis first or last, so as to avoid touching the coverslip*)

`reference_move_not_Z`(*r*, *safe=False*)

Moves the unit to position r in reference camera system, without moving the stage, but without moving the Z axis (so this can be done last).

Parameters

- **r** (*XYZ position vector in um*)
- **safe** (*if True, moves the Z axis first or last, so as to avoid touching the coverslip*)

`reference_position`()

Position in the reference camera system.

Return type
The current position in um as an XYZ vector.

`reference_relative_move`(*r*)

Moves the unit by vector r in reference camera system, without moving the stage.

Parameters
r (*XYZ position vector in um*)

`refine`()

Refine the calibration by iterating over large movements.

`safe_move`(*r*, *withdraw=0.0*, *recalibrate=False*)

Moves the device to position x (an XYZ vector) in a way that minimizes interaction with tissue.

If the movement is down, the manipulator is first moved horizontally, then along the pipette axis. If the movement is up, a direct move is done.

Parameters

- **r** (*target position in um, an (X,Y,Z) vector*)
- **withdraw** (*in um; if not 0, the pipette is withdrawn by this value from the target position x*)
- **recalibrate** (*if True, pipette is recalibrated 1 mm before its target*)

`save_configuration`()

Outputs configuration in a dictionary.

`save_state`()

Save the current state (e.g. the position of the manipulators) for later recovery in the case of a failure or abort. Has to be overwritten in subclasses. Should save the state to the `saved_state` variable or overwrite `has_saved_state` as well.

`take_photos`(*rig=1*)

Take photos of the pipette. It is assumed that the pipette is centered and in focus.

`withdraw`()

Withdraw the pipette to the upper end position

exception holypipette.devices.manipulator.calibratedunit.**CalibrationError**(*message='Device is not calibrated'*)

Bases: Exception

holypipette.devices.manipulator.fakemanipulator module

A fake device useful for development. It has 9 axes, numbered 1 to 9.

class holypipette.devices.manipulator.fakemanipulator.**FakeManipulator**(*min=None*, *max=None*, *angle=25.0*)

Bases: *Manipulator*

absolute_move(*x*, *axis*)

Moves the device axis to position x.

Parameters

- **axis** (*axis number*)
- **x** (*target position in um.*)

position(*axis*)

Current position along an axis.

Parameters
axis (*axis number*)

Return type
The current position of the device axis in um.

holypipette.devices.manipulator.leica module

A Z Unit for a Leica microscope, using MicroManager. Communication through serial COM port.

class holypipette.devices.manipulator.leica.**Leica**(*name='COM1'*)

Bases: *Microscope*

absolute_move(*x*)

Moves the device axis to position x in um.

Parameters

- **axis** (*this is ignored*)
- **x** (*target position in um.*)

position()

Current position along an axis.

Parameters
axis (*this is ignored*)

Return type
The current position of the device axis in um.

relative_move(*x*)

Moves the device axis by relative amount x in um.

Parameters

- **axis** (*this is ignored*)
- **x** (*position shift in um.*)

step_move(*distance*)

stop()

Stop current movements.

wait_until_still()

Waits for the motors to stop.

holypipette.devices.manipulator.luigsneumann_SM10 module

Manipulator class for the Luigs and Neumann SM-10 manipulator controller.

Adapted from Michael Graupner's LandNSM5 class.

Not all commands are implemented.

class holypipette.devices.manipulator.luigsneumann_SM10.**LuigsNeumann_SM10**(*name=None*, *stepmoves=True*)

Bases: *SerialDevice*, *Manipulator*

absolute_move(*x*, *axis*, *fast=None*)

Moves the device axis to position x.

Parameters

- **axis** (*axis number (starting at 1)*)
- **x** (*target position in um.*)
- **speed** (*optional speed in um/s.*)
- **fast** (*True if fast move, False if slow move.*)

absolute_move_group(*x*, *axes*, *fast=None*)

Moves the device group of axes to position x.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*target position in um (vector or list)*)
- **fast** (*True if fast move, False if slow move.*)

fast_speed(*axis*)

Queries the fast speed setting for a given axis

go_to_zero(*axes*)

Moves axes to zero position.

home(*axis*)

Move the axis to home.

home_abort(*axis*)

Aborts home movement.

home_return(*axis*)

Returns to position before home command.

position(*axis*)

Current position along an axis.

Parameters
axis (*axis number (starting at 1)*)

Return type
The current position of the device axis in um.

position2(*axis*)

Current position along an axis, using the second counter.

Parameters
axis (*axis number (starting at 1)*)

Return type
The current position of the device axis in um.

position_group(*axes*)

Current position along a group of axes.

Parameters
axes (*list of axis numbers*)

Return type
The current position of the device axis in um (vector).

relative_move(*x*, *axis*, *fast=None*)

Moves the device axis by relative amount x in um.

Parameters

- **axis** (*axis number*)
- **x** (*position shift in um.*)
- **fast** (*True if fast move, False if slow move. None: decide based on distance.*)

relative_move_group(*x*, *axes*, *fast=None*)

Moves the device group of axes by relative amount x in um.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*position shift in um (vector or list).*)
- **fast** (*True if fast move, False if slow move. None: decide based on distance.*)

send_command(*ID*, *data*, *nbytes_answer*)

Send a command to the controller

set_fast_speed(*axis*, *speed*)

Sets the fast speed setting for a given axis

set_home_direction(*axis*, *direction*)

Sets home direction.

set_home_velocity(*axis*, *velocity*)

Sets home direction. Velocity between 0 and 15.

set_ramp_length(*axis*, *length*)

Sets the ramp length for the chosen axis

Parameters

- **axis** (*axis number*)
- **length** (*length between 0 and 16*)

set_single_step_distance(*axis*, *distance*)

Distance (in um) for *single_step*.

set_single_step_factor_trackball(*axis*, *factor*)

Sets the single step factor with the trackball command

Parameters

- **axis** (*axis number*)
- **factor** (*single step factor (what is it ??)*)

set_single_step_velocity(*axis*, *velocity*)

Velocity for *single_step*. See table rps_slow.

set_slow_speed(*axis*, *speed*)

Sets the slow speed setting for a given axis

single_step(*axis*, *steps*)

Moves the given axis by a signed number of steps using the StepIncrement or StepDecrement command. Using a steps argument different from 1 (or -1) simply sends multiple StepIncrement/StepDecrement commands. Uses distance and velocity set by *set_single_step_distance* resp. *set_single_step_velocity*.

single_step_trackball(*axis*, *steps*)

Makes a number of single steps with the trackball command

Parameters

- **axis** (*axis number*)
- **steps** (*number of steps*)

slow_speed(*axis*)

Queries the slow speed setting for a given axis

step_move(*distance*, *axis=None*, *maxstep=255*)

Relative move using steps of up to 255 um. This fixes a bug on L&N controller.

stop(*axis*)

Stops current movements on one axis.

stop_all()

Stops all 9 axes (could be more).

wait_until_still(*axes=None*)

Waits for the motors to stop. On SM10, commands of motors seem to block.

zero(*axes*)

Sets the current position of the axes as the zero position.

zero2(*axes*)

Sets the current position of the axes as the zero position on the second counter.

holypipette.devices.manipulator.luigsneumann_SM5 module

Manipulator class for the Luigs and Neumann SM-5 manipulator controller.

Adapted from Michael Graupner's LandNSM5 class.

Not all commands are implemented.

class holypipette.devices.manipulator.luigsneumann_SM5.**LuigsNeumann_SM5**(*name=None*, *stepmoves=True*)

Bases: *SerialDevice*, *Manipulator*

absolute_move(*x*, *axis*)

Moves the device axis to position x. It uses the fast movement command.

Parameters

- **axis** (*axis number (starting at 1)*)
- **x** (*target position in um.*)
- **speed** (*optional speed in um/s.*)

absolute_move_group(*x*, *axes*)

Moves the device group of axes to position x.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*target position in um (vector or list).*)

establish_connection()

go_to_zero(*axes*)

Moves axes to zero position.

position(*axis*)

Current position along an axis.

Parameters
axis (*axis number (starting at 1)*)

Return type
The current position of the device axis in um.

position2(*axis*)

Current position along an axis on the second counter.

Parameters
axis (*axis number (starting at 1)*)

Return type
The current position of the device axis in um.

relative_move(*x*, *axis*)

Moves the device axis by relative amount x in um. It uses the fast command.

Parameters

- **axis** (*axis number*)
- **x** (*position shift in um.*)

send_command(*ID*, *data*, *nbytes_answer*, *ack_ID=''*, *resends=0*)

Send a command to the controller

set_ramp_length(*axis*, *length*)

Set the ramp length for the chosen axis :param axis: axis which ramp shall be changed :param length: 0<length<=16 :return:

set_single_step_distance(*axis*, *distance*)

Distance (in um) for *single_step*.

set_to_zero_second_counter(*axes*)

Sets the current position of the axes as the zero position on the second counter.

single_step(*axis*, *steps*)

Moves the given axis by a signed number of steps using the StepIncrement or StepDecrement command. Using a steps argument different from 1 (or -1) simply sends multiple StepIncrement/StepDecrement commands. Uses distance and velocity set by *set_single_step_distance* resp. `set_single_step_velocity`.

step_move(*distance*, *axis=None*, *maxstep=255*)

Relative move using steps of up to 255 um. This fixes a bug on L&N controller.

stop(*axis*)

Stop current movements.

wait_until_still(*axes=None*)

Waits for the motors to stop.

zero(*axes*)

Sets the current position of the axes as the zero position.

holypipette.devices.manipulator.manipulator module

Generic Manipulator class for manipulators.

To make a new device, one must implement at least: * position * absolute_move

TODO: * Add minimum and maximum for each axis

class holypipette.devices.manipulator.manipulator.**Manipulator**

Bases: *TaskController*

absolute_move(*x*, *axis=None*)

Moves the device axis to position x.

Parameters

- **axis** (*axis number*)
- **x** (*target position in um.*)

absolute_move_group(*x*, *axes*)

Moves the device group of axes to position x.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*target position in um (vector or list).*)

`delete_state()`

Delete any previously saved state. By default, overwrites the `saved_state` attribute with `None`.

`position`(*axis=None*)

Current position along an axis.

Parameters
axis (*axis number*)

Return type
The current position of the device axis in um.

`position_group`(*axes*)

Current position along a group of axes.

Parameters
axes (*list of axis numbers*)

Return type
The current position of the device axis in um (vector).

`recover_state()`

Recover the state (e.g. the position of the manipulators) after a failure or abort. Has to be overwritten in subclasses.

`relative_move`(*x*, *axis*)

Moves the device axis by relative amount x in um.

Parameters

- **axis** (*axis number*)
- **x** (*position shift in um.*)

`relative_move_group`(*x*, *axes*)

Moves the device group of axes by relative amount x in um.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*position shift in um (vector or list).*)

`save_state()`

Save the current state (e.g. the position of the manipulators) for later recovery in the case of a failure or abort. Has to be overwritten in subclasses. Should save the state to the `saved_state` variable or overwrite `has_saved_state` as well.

`stop`(*axis*)

Stops current movements.

`wait_until_reached`(*position*, *axes=None*, *precision=0.5*, *timeout=10*)

Waits until position is reached within precision, and raises an error if the target is not reached after the time out, unless the manipulator is still moving.

Parameters

- **position** (*target position in micrometer*)
- **axes** (*axis number of list of axis numbers*)
- **precision** (*precision in micrometer*)

- **timeout** (*time out in second*)

wait_until_still(*axes=None*)

Waits until motors have stopped.

Parameters
axes (*list of axis numbers*)

exception holypipette.devices.manipulator.manipulator.**ManipulatorError**(*message='Device is not calibrated'*)

Bases: Exception

holypipette.devices.manipulator.manipulatorunit module

A class for access to a particular unit managed by a device. It is essentially a subset of a Manipulator

class holypipette.devices.manipulator.manipulatorunit.**ManipulatorUnit**(*dev*, *axes*)

Bases: *Manipulator*

absolute_move(*x*, *axis=None*)

Moves the device axis to position x in um.

Parameters

- **axis** (*axis number starting at 0; if None, all XYZ axes*)
- **x** (*target position in um.*)

absolute_move_group(*x*, *axes*)

Moves the device group of axes to position x.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*target position in um (vector or list).*)

is_accessible(*x*, *axis=None*)

Checks whether position x is accessible.

THIS METHOD IS INCORRECT.

motor_ranges()

Runs the motors to calculate ranges of the motors.

DOESN'T WORK! DO NOT USE!

position(*axis=None*)

Current position along an axis.

Parameters
axis (*axis number starting at 0; if None, all XYZ axes*)

Return type
The current position of the device axis in um.

relative_move(*x*, *axis=None*)

Moves the device axis by relative amount x in um.

Parameters

- **axis** (*axis number starting at 0; if None, all XYZ axes*)

- **x** (*position shift in um.*)

stop(*axis=None*)

Stop current movements.

wait_until_reached(*position*, *axes=None*, *precision=0.5*, *timeout=10*)

Waits until position is reached within precision, and raises an error if the target is not reached after the time out, unless the manipulator is still moving.

Parameters

- **position** (*target position in micrometer*)
- **axes** (*axis number of list of axis numbers*)
- **precision** (*precision in micrometer*)
- **timeout** (*time out in second*)

wait_until_still(*axes=None*)

Waits for the motors to stop.

holypipette.devices.manipulator.microscope module

A microscope is a manipulator with a single axis. With methods to take a stack of images, autofocus, etc.

TODO: * a umanager class that autoconfigures with umanager config file * steps for stack acquisition?

class holypipette.devices.manipulator.microscope.**Microscope**(*dev*, *axis*)

Bases: *Manipulator*

A microscope Z axis, obtained here from an axis of a Manipulator.

absolute_move(*x*)

Moves the device axis to position x in um.

Parameters

x (*target position in um.*)

load_configuration(*config*)

Loads configuration from dictionary config. Variables not present in the dictionary are untouched.

position()

Current position

Return type

The current position of the device axis in um.

relative_move(*x*)

Moves the device axis by relative amount x in um.

Parameters

x (*position shift in um.*)

save_configuration()

Outputs configuration in a dictionary.

stack(*camera*, *z*, *preprocessing=<function Microscope.<lambda>>*, *save=None*, *pause=0.3*)

Take a stack of images at the positions given in the z list

Parameters

- **camera** (*a camera, eg with a snap() method*)
- **z** (*A list of z positions*)
- **preprocessing** (*a function that processes the images (optional)*)
- **save** (*saves images to disk if True*)
- **pause** (*pause in second after each movement*)

`step_move`(*distance*)

`stop`()

Stop current movements.

`wait_until_still`()

Waits for the motors to stop.

holypipette.devices.manipulator.proscan module

Prior Proscan III Stage control.

`class holypipette.devices.manipulator.proscan.Prior`

Bases: *Manipulator*

`absolute_move`(*x*, *axis*)

Moves the device axis to position x.

Parameters

- **axis** (*axis number*)
- **x** (*target position in um.*)

`absolute_move_group`(*x*, *axes*)

Moves the device group of axes to position x.

Parameters

- **axes** (*list of axis numbers*)
- **x** (*target position in um (vector or list).*)

`position`(*axis*)

Current position along an axis.

Parameters
axis (*axis number*)

Return type
The current position of the device axis in um.

`position_group`(*axes*)

Current position along a group of axes.

Parameters
axes (*list of axis numbers*)

Return type
The current position of the device axis in um (vector).

relative_move(*x*, *axis*)

Moves the device axis by relative amount x in um.

Parameters

- **axis** (*axis number*)
- **x** (*position shift in um.*)

stop()

Stops current movements.

wait_until_still(*axes=None*, *axis=None*)

Waits until motors have stopped.

Parameters
axes (*list of axis numbers*)

holypipette.devices.manipulator.sensapex module

holypipette.devices.pressurecontroller package

Submodules

holypipette.devices.pressurecontroller.ob1 module

Elveflow OB1 microfluidic flow control system

Running this program calibrates the pressure controller.

class holypipette.devices.pressurecontroller.ob1.**OB1**(*calibrate=False*)

Bases: *PressureController*

measure(*port=0*)

Measures the instantaneous pressure, on designated port.

set_pressure(*pressure*, *port=0*)

Sets the pressure, on designated port.

holypipette.devices.pressurecontroller.pressurecontroller module

A general pressure controller class

class holypipette.devices.pressurecontroller.pressurecontroller.**FakePressureController**

Bases: *PressureController*

get_pressure(*port=0*)

Gets the pressure on the designated port. Note that this does not refer to any measurement, but simply to the pressure as set via *set_pressure*.

measure(*port=0*)

Measures the instantaneous pressure, on designated port.

set_pressure(*pressure*, *port=0*)

Sets the pressure, on designated port.

class holypipette.devices.pressurecontroller.pressurecontroller.**PressureController**

Bases: *TaskController*

get_pressure(*port=0*)

Gets the pressure on the designated port. Note that this does not refer to any measurement, but simply to the pressure as set via *set_pressure*.

measure(*port=0*)

Measures the instantaneous pressure, on designated port.

ramp(*amplitude=- 230.0*, *duration=1.5*, *port=0*)

Makes a ramp of pressure

set_pressure(*pressure*, *port=0*)

Sets the pressure, on designated port.

Submodules

holypipette.devices.serialdevice module

The SerialDevice class: a device that communicates through the serial port.

class holypipette.devices.serialdevice.**SerialDevice**(*name=None*)

Bases: object

A device that communicates through the serial port.

CRC_16(*butter*, *length*)

holypipette.geometry package

Submodules

holypipette.geometry.planes module

Calculation related to planes (intersections, etc)

class holypipette.geometry.planes.**Plane**(*vector*, *offset*)

Bases: object

A plane is defined by its normal vector n and an offset a, according to: n.x + a = 0

static from_points(*x1*, *x2*, *x3*)

Returns a plane determined by three points

parallel_plane(*x*)

Returns a parallel plane passing by x

project(*x*, *u=None*)

Projects point x to the plane along vector u. If u is not specified, default is the normal vector, i.e., orthogonal projection

signed_distance(*x*, *u*)

Signed distance of x from the plane along vector u. That is, returns k such that x + k.u is in the plane.

holypipette.gui package

Submodules

holypipette.gui.camera module

class holypipette.gui.camera.**CameraGui**(*camera*, *image_edit=None*, *display_edit=None*, *with_tracking=False*, *base_directory='.'*)

Bases: QMainWindow

The basic GUI for showing a camera image.

Parameters

- **camera** (Camera) – The Camera object that will be used for displaying an image via *LiveFeedQt*.
- **image_edit** (*function or list of functions, optional*) – A function that will be called with the numpy array returned by the camera. Can be used to post-process the image, e.g. to change its brightness.
- **display_edit** (*function or list of functions, optional*) – A function that will be called with the QPixmap that is based on the camera image. Can be used to display additional information on top of this image, e.g. a scale bar or text.
- **with_tracking** (*bool, optional*) – Whether to activate the object tracking interface. Defaults to False.

abort_task()

add_config_gui(*config*)

camera_reset_signal

camera_signal

close()

Close the GUI.

closeEvent(*self*, *QCloseEvent*)

configuration_keypress()

Show/hide the configuration pane

display_edit(*pixmap*)

Applies the functions stored in display_edit_funcs to the video image pixmap.

Parameters

pixmap (QPixmap) – The pixmap to draw on.

draw_cross(*pixmap*)

Draws a cross at the center. Meant to be used as a display_edit function.

Parameters

pixmap (QPixmap) – The pixmap to draw on.

error_status(*message*)

exit()

Exit the application

help_keypress()

Toggle display of keyboard/mouse commands

image_edit(*image*)

Applies the functions stored in `image_edit_funcs` to the video image. Each function works on the result of the previous function

Parameters
image (ndarray) – The original video image or the image returned by a previously called function.

Returns
new_image – The post-processed image. Should be of the same size and data type as the original image.

Return type
ndarray

initialize()

keyPressEvent(*self*, *QKeyEvent*)

log_keypress()

Toggle display of log output

log_signal

register_commands()

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

register_key_action(*key*, *modifier*, *command*, *argument=None*, *default_doc=True*)

Link a keypress to an action.

Parameters

- **key** (`Qt.Key`) – The key that should be handled, specified as a `Qt` constant, e.g. `Qt.Key_X` or `Qt.Key_5`.
- **modifier** (`Qt.Modifer` or `None`) – The modifier that needs to be pressed at the same time to trigger the action. The modifier needs to be given as a `Qt` constant, e.g. `Qt.ShiftModifier` or `Qt.ControlModifier`. Alternatively, `None` can be used to specify that the keypress should lead to the action independent of the modifier.
- **command** (*method*) – A method implementing the action that has been annotated with the *@command* or *@blocking_command* decorator.
- **argument** (*object, optional*) – An additional argument that should be handled to the method defined as `command`. Can be used to re-use the same action in a parametrized way (e.g. steps of different size).
- **default_doc** (*bool, optional*) – Whether to include the action in the automatically generated help. Defaults to `True`.

register_mouse_action(*click_type*, *modifier*, *command*, *default_doc=True*)

Link a mouse click on the camera image to an action.

Parameters

- **click_type** (`Qt.MouseButton`) – The type of click that should be handled as a `Qt` constant, e.g. `Qt.LeftButton` or `Qt.RightButton`.
- **modifier** (`Qt.Modifer` or `None`) – The modifier that needs to be pressed at the same time to trigger the action. The modifier needs to be given as a `Qt` constant, e.g. `Qt.ShiftModifier` or `Qt.ControlModifier`. Alternatively, `None` can be used to specify that the mouse click should lead to the action independent of the modifier.
- **command** (*method*) – A method implementing the action that has been annotated with the `@command` or `@blocking_command` decorator.
- **default_doc** (*bool, optional*) – Whether to include the action in the automatically generated help. Defaults to `True`.

set_status_message(*category*, *message*)

splitter_size_changed(*pos*, *index*)

start_task(*task_name*, *interface*)

status_message_updated(*message*)

task_finished(*exit_reason*, *controller_or_message*)

toggle_configuration_display()

toggle_help()

toggle_log()

toggle_overlay()

Show/hide the overlay information on the image

toggle_recording(**args*)

Toggle recording image files to disk

video_mouse_press(*event*)

class holypipette.gui.camera.**ConfigGui**(*config*, *show_name=False*)

Bases: `QWidget`

display_changed_value(*key*, *value*)

load_config()

save_config()

set_boolean_value(*name*, *widget*)

set_numerical_value(*name*, *value*)

set_numerical_value_with_unit(*name*, *magnitude*, *value*)

value_changed(*key*, *value*)

value_changed_signal

class holypipette.gui.camera.**ElidedLabel**(*text*, *minimum_width=200*, **args*, ***kwds*)

Bases: QLabel

minimumSizeHint(*self*) → QSize

resizeEvent(*self*, *QResizeEvent*)

class holypipette.gui.camera.**KeyboardHelpWindow**(*parent*)

Bases: QMainWindow

closeEvent(*self*, *QCloseEvent*)

close_signal

keyPressEvent(*self*, *QKeyEvent*)

register_custom_action(*category*, *action*, *description*)

register_key_action(*key*, *modifier*, *category*, *description*)

register_mouse_action(*click_type*, *modifier*, *category*, *description*)

update_text()

class holypipette.gui.camera.**LogNotifyHandler**(*signal*)

Bases: Handler

emit(*record*)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class holypipette.gui.camera.**LogViewerWindow**(*parent*)

Bases: QMainWindow

closeEvent(*self*, *QCloseEvent*)

close_signal

levels = {'DEBUG': 10, 'ERROR': 40, 'INFO': 20, 'WARN': 30}

save_log()

set_level(*level_idx*)

class holypipette.gui.camera.**Logger**

Bases: QAbstractTableModel, Handler

columnCount(*self*, *parent: QModelIndex = QModelIndex()*) → int

data(*self*, *QModelIndex*, *role: int = Qt.DisplayRole*) → Any

emit(*record*)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

headerData(*self*, *int*, *Qt.Orientation*, *role: int = Qt.DisplayRole*) → Any

rowCount(*self*, *parent: QModelIndex = QModelIndex()*) → int

save_to_file(*filename*)

class holypipette.gui.camera.**RecordingDialog**(*base_directory*, *frame_rate*, *pixels*, *settings*, *parent=None*)

Bases: QDialog

directory_clicked()

memory_edited(*value*)

prefix_edited()

select_folder()

skip_edited(*value*)

holypipette.gui.livefeed module

class holypipette.gui.livefeed.**LiveFeedQt**(*camera*, *image_edit=None*, *display_edit=None*, *mouse_handler=None*, *parent=None*)

Bases: QLabel

mousePressEvent(*self*, *QMouseEvent*)

update_image()

holypipette.gui.manipulator module

class holypipette.gui.manipulator.**ManipulatorGui**(*camera*, *pipette_interface*, *with_tracking=False*)

Bases: *CameraGui*

display_manipulator(*pixmap*)

Displays the number of the selected manipulator.

display_timer(*pixmap*)

draw_scale_bar(*pixmap*, *text=True*, *autoscale=True*, *position=True*)

measure_ranges()

Measure manipulator ranges

pipette_command_signal

pipette_reset_signal

register_commands(*manipulator_keys=True*)

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

show_tip(*pixmap*)

show_tip_switch()

Show the tip of selected manipulator

holypipette.gui.movingList module

holypipette.gui.paramecium_device module

GUI for Paramecium electrophysiology

class holypipette.gui.paramecium_device.**ParameciumDeviceGui**(*camera*, *pipette_interface*)

Bases: *ManipulatorGui*

paramecium_command_signal

paramecium_reset_signal

register_commands()

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

holypipette.gui.paramecium_device2 module

Simplified Paramecium device GUI

GUI for Paramecium electrophysiology, with the immobilization device

class holypipette.gui.paramecium_device2.**ParameciumDeviceGui**(*stage*, *microscope*, *camera*, *units*, *config_filename='paramecium_device.cfg'*)

Bases: *ManipulatorGui*

register_commands()

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

holypipette.gui.paramecium_droplet module

GUI for Paramecium electrophysiology

class holypipette.gui.paramecium_droplet.**ParameciumDropletGui**(*camera*, *pipette_interface*)

Bases: *ManipulatorGui*

paramecium_command_signal

paramecium_reset_signal

register_commands()

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

show_paramecium(*pixmap*)

track_paramecium(*frame*)

holypipette.gui.paramecium_droplet.**create_painter**(*pixmap*, *color*, *width=1*)

Setup a QPainter with a QPen of a given color and width.

Parameters

- **pixmap** (QPixMap) – The pixmap on which to draw.
- **color** (*tuple*) – The 4-element tuple defining the color (R, G, B, alpha).
- **width** (*int*) – The width in pixels.

Returns
painter – The painter that can be used for drawing on the pixmap.

Return type
QPainter

holypipette.gui.paramecium_droplet.**draw_contour**(*contour*, *painter*, *scale*)

holypipette.gui.paramecium_droplet.**draw_ellipse**(*painter*, *x*, *y*, *width*, *height*, *angle*, *pixel_per_um*, *scale*)

holypipette.gui.patch module

class holypipette.gui.patch.**PatchGui**(*camera*, *pipette_interface*, *patch_interface*, *with_tracking=False*)

Bases: *ManipulatorGui*

display_pressure()

patch_command_signal

patch_reset_signal

register_commands()

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

class holypipette.gui.patch.**TrackingPatchGui**(*camera*, *pipette_interface*, *patch_interface*, *with_tracking=False*)

Bases: *PatchGui*

register_commands()

Tie keypresses and mouse clicks to commands. Should call *register_key_action* and *register_mouse_action*. Overriding methods in subclass should call the superclass if they want to keep the commands registered by the superclass(es).

holypipette.interface package

Package defining *TaskInterface* classes. These classes are the interfaces between the GUI and the classes that perform the actual tasks such as patching, controlling the camera, etc. The key role of the *TaskInterface* classes is to define the commands that it supports (e.g. patching, moving the manipulators, etc.) and what should be done if such command is received.

Submodules

holypipette.interface.base module

Package defining the *TaskInterface* class, central to the interface between GUI and *TaskController* objects.

class holypipette.interface.base.**TaskInterface**

Bases: QObject, *LoggingObject*

Class defining the basic interface between the GUI and the objects controlling the hardware. Classes inheriting from this class should:

- Call this class's __init__ function in its __init__
- Annotate all functions providing commands with the *@command* or *@blocking_command* decorator.
- To correctly interact with the GUI for blocking commands (show that task is running, show error message if task fails, etc.), the method needs to call the *execute* function to execute the command.

abort_task()

The user asked for an abort of the currently running (blocking) command. We transmit this information to all executing objects (for simplicity, only one should be running) by setting the TaskController.abort_requested attribute. The object runs in a separate thread, but will finish its operation as soon as it checks for this attribute (either by explicitly checking with *TaskController.abort_if_requested*, or by using *TaskController.sleep* or one of the logging methods).

command_received(*command*, *argument*)

Slot that is triggered when the GUI triggers a command handled by this *TaskInterface*. If an error occurs in the handling of the command (e.g., the command does not exist or received the wrong number of arguments), an error is logged and the *task_finished* signal is emitted. Note that the handling of errors *within* the command, as well as the handling of abort requests is performed in the *execute* method.

Parameters

- **command** (*method*) – A reference to the requested command.
- **argument** (*object*) – The argument of the requested command (possibly None).

connect(*main_gui*)

Connect signals to slots in the main GUI. Will be called automatically during initialization of the GUI.

Parameters
main_gui (*CameraGui*) – The main GUI in control.

execute(*task*, *argument=None*)

Execute a function in a *TaskController* and signal the (successful or unsuccessful) completion via the *task_finished* signal.

Can either execute a single task or a chain of tasks where each task is only executed when the previous was successful.

Parameters

- **task** (*method or list of methods*) – A method of a TaskController object that should be executed, or a list of such methods.
- **argument** (*object or list of object, optional*) – An argument that will be provided to task or None (the default). For a chain of function calls, provide a list of arguments.

Returns

success – Whether the execution was completed successfully. This can be used to manually enchain multiple tasks to avoid calling subsequent tasks after a failed/aborted task. Note that it can be easier to pass a list of functions instead.

Return type

bool

reset_requested(*controller*)

Slot that will be triggered when the user asks for resetting the state after an aborted or failed command.

Parameters

controller (*TaskController*) – The object that was executing the task that failed or was aborted. This object is requested to reset its state.

task_finished

Signals the end of a task with an "error code": 0: successful execution; 1: error during execution; 2: aborted

holypipette.interface.base.**blocking_command**(*category*, *description*, *task_description*, *default_arg=None*)

Decorator that annotates a function with information about the implemented (blocking) command.

Parameters

- **category** (*str*) – The command category (used for structuring the help window).
- **description** (*str*) – A descriptive text for the command (used in the help window).
- **task_description** (*str*) – Text that will be displayed to the user while the task is running
- **default_arg** (*object, optional*) – A default argument provided to the method or `None` (the default).

holypipette.interface.base.**command**(*category*, *description*, *default_arg=None*, *success_message=None*)

Decorator that annotates a function with information about the implemented command.

Parameters

- **category** (*str*) – The command category (used for structuring the help window).
- **description** (*str*) – A descriptive text for the command (used in the help window).
- **default_arg** (*object, optional*) – A default argument provided to the method or `None` (the default).
- **success_message** (*str, optional*) – A message that will be displayed in the status bar of the GUI window after the execution of the command. For simple commands that have visual feedback, e.g. moving the manipulator or changing the exposure time, this should not be set to avoid unnecessary messages. For actions that have no visual feedback, e.g. storing a position, this should be set to give the user an indication that something happened.

holypipette.interface.camera module

class holypipette.interface.camera.**CameraInterface**(*camera*, *with_tracking=False*)

Bases: *TaskInterface*

auto_exposure(*args*)

Auto exposure

Parameters
args (*object, optional*)

connect(*main_gui*)

Connect signals to slots in the main GUI. Will be called automatically during initialization of the GUI.

Parameters
main_gui (*CameraGui*) – The main GUI in control.

decrease_exposure(*decrease*)

Decrease exposure time by 2.5ms

Parameters
decrease (*object, optional*) – If no argument is given, 2.5 will be used as a default argument

increase_exposure(*increase*)

Increase exposure time by 2.5ms

Parameters
increase (*object, optional*) – If no argument is given, 2.5 will be used as a default argument

pipette_contact_detection(*img*)

save_image()

Save the current image to a file

show_tracked_objects(*img*)

show_tracked_paramecium(*img*)

signal_updated_exposure()

track_object(*position=None*)

Select an object for automatic tracking

Parameters
position (*object, optional*)

updated_exposure

holypipette.interface.paramecium_device module

class holypipette.interface.paramecium_device.**CalibratedUnitProxy**(*pipette_interface*)

Bases: object

Small helper object that forwards all requests to the currently selected manipulator.

class holypipette.interface.paramecium_device.**ParameciumDeviceConfig**(*value_changed=None*, **args*, ***kwds*)

Bases: *Config*

params(calibration_level=NumberWithUnit, impalement_level=NumberWithUnit, impalement_step=NumberWithUnit, pause_between_steps=NumberWithUnit, pipette_distance=NumberWithUnit, short_withdraw_distance=NumberWithUnit, withdraw_distance=NumberWithUnit, working_level=NumberWithUnit, name=String) [1;32mParameters of 'ParameciumDeviceConfig' ==========[0m [1;31mParameters changed from their default values are marked in red.[0m [1;36mSoft bound values are marked in cyan.[0m C/V= Constant/Variable, RO/RW = ReadOnly/ReadWrite, AN=Allow None

[1;34mName Value Type Bounds Mode [0m

working_level 50 NumberWithUnit (0, 500) V RW calibration_level 200 NumberWithUnit (0, 1000) V RW impalement_level 10 NumberWithUnit (0, 100) V RW withdraw_distance 1000 NumberWithUnit (0, 3000) V RW pipette_distance 250 NumberWithUnit (0, 2000) V RW short_withdraw_distance 20 NumberWithUnit (0, 100) V RW impalement_step 5 NumberWithUnit (1, 10) V RW pause_between_steps 0.5 NumberWithUnit (0, 2) V RW

[1;32mParameter docstrings: ==========[0m

[1;34mworking_level: Working level[0m [1;31mcalibration_level: Calibration level[0m [1;34mimpalement_level: Impalement level[0m [1;31mwithdraw_distance: Withdraw distance[0m [1;34mpipette_distance: Pipette distance from center[0m [1;31mshort_withdraw_distance: Withdraw before impalement[0m [1;34mimpalement_step: Step size for impalement[0m [1;31mpause_between_steps: Pause between impalement steps[0m

calibration_level = 200

categories = [('Manipulation', ['working_level', 'calibration_level', 'impalement_level', 'withdraw_distance', 'pipette_distance', 'short_withdraw_distance']), ('Automation', ['impalement_step', 'pause_between_steps'])]

impalement_level = 10

impalement_step = 5

name = 'ParameciumDeviceConfig'

oscilloscope_filename = '/home/docs/holypipette/oscilloscope.txt'

pause_between_steps = 0.5

pipette_distance = 250

short_withdraw_distance = 20

withdraw_distance = 1000

working_level = 50

class holypipette.interface.paramecium_device.**ParameciumDeviceInterface**(*pipette_interface*, *camera*)

Bases: *TaskInterface*

focus_calibration_level()

Focus on calibration level

focus_working_level()

Focus on working level

move_pipette_down()

Move pipette vertically to impalement level

move_pipette_in()

Move pipette to impalement level by a side move

move_pipette_until_drop()

Move pipette down until potential drop

move_pipette_working_level(*xy_position*)

Move pipette down to position at working distance level

Parameters
xy_position (*object, optional*)

partial_withdraw()

Partially withdraw the pipette

class holypipette.interface.paramecium_device.**ParameciumDeviceSimplifiedInterface**(*stage*, *microscope*, *camera*, *units*, *config_filename=None*)

Bases: *PipetteInterface*

autocenter()

Center the stage below the objective

focus_calibration_level()

Focus on calibration level

focus_working_level()

Focus on working level

move_pipette_down()

Move pipette vertically to impalement level

move_pipette_in()

Move pipette to impalement level by a side move

move_pipette_until_drop()

Move pipette down until potential drop

move_pipette_working_level(*xy_position*)

Move pipette down to position at working distance level

Parameters
xy_position (*object, optional*)

partial_withdraw()

Partially withdraw the pipette

holypipette.interface.paramecium_droplet module

class holypipette.interface.paramecium_droplet.**CalibratedUnitProxy**(*pipette_interface*)

Bases: object

Small helper object that forwards all requests to the currently selected manipulator.

class holypipette.interface.paramecium_droplet.**ParameciumDropletConfig**(*value_changed=None*, **args*, ***kwds*)

Bases: *Config*

params(autofocus_size=NumberWithUnit, autofocus_sleep=NumberWithUnit, blur_size=NumberWithUnit, draw_contours=Boolean, draw_fitted_ellipses=Boolean, max_displacement=NumberWithUnit, max_gradient=NumberWithUnit, max_length=NumberWithUnit, max_width=NumberWithUnit, min_gradient=NumberWithUnit, min_length=NumberWithUnit, min_width=NumberWithUnit, minimum_contour=NumberWithUnit, minimum_stop_time=NumberWithUnit, stop_amplitude=NumberWithUnit, stop_duration=NumberWithUnit, target_pixelperum=Number, working_distance=NumberWithUnit, name=String) [1;32mParameters of 'ParameciumDropletConfig' ==========[0m [1;31mParameters changed from their default values are marked in red.[0m [1;36mSoft bound values are marked in cyan.[0m C/V= Constant/Variable, RO/RW = ReadOnly/ReadWrite, AN=Allow None

[1;34mName Value Type Bounds Mode [0m

target_pixelperum 1 Number (0, 4) V RW min_gradient 75 NumberWithUnit (0, 100) V RW max_gradient 98 NumberWithUnit (0, 100) V RW blur_size 10 NumberWithUnit (0, 100) V RW minimum_contour 100 NumberWithUnit (0, 1000) V RW min_length 65 NumberWithUnit (0, 1000) V RW max_length 170 NumberWithUnit (0, 1000) V RW min_width 30 NumberWithUnit (0, 1000) V RW max_width 60 NumberWithUnit (0, 1000) V RW max_displacement 50 NumberWithUnit (0, 1000) V RW autofocus_size 150 NumberWithUnit (0, 1000) V RW autofocus_sleep 0.5 NumberWithUnit (0, 1) V RW minimum_stop_time 0 NumberWithUnit (0, 5000) V RW stop_duration 50 NumberWithUnit (0, 1000) V RW stop_amplitude 5 NumberWithUnit (0, 1000) V RW working_distance 200 NumberWithUnit (0, 1000) V RW draw_contours False Boolean (0, 1) V RW draw_fitted_ellipses False Boolean (0, 1) V RW

[1;32mParameter docstrings: ==========[0m

[1;34mtarget_pixelperum: Target number of pixel per um[0m [1;31mmin_gradient: Minimum gradient quantile for edge detection[0m [1;34mmax_gradient: Maximum gradient quantile for edge detection[0m [1;31mblur_size: Gaussian blurring size[0m [1;34mminimum_contour: Minimum contour length[0m [1;31mmin_length: Minimum length ellipsis[0m [1;34mmax_length: Maximum length for ellipsis[0m [1;31mmin_width: Minimum width for ellipsis[0m [1;34mmax_width: Maximum width for ellipsis[0m [1;31mmax_displacement: Maximum displacement over one frame[0m [1;34mautofocus_size: Size of bounding box for autofocus[0m [1;31mautofocus_sleep: Sleep time autofocus[0m [1;34mminimum_stop_time: Time before starting automation[0m [1;31mstop_duration: Stopping duration before detection[0m [1;34mstop_amplitude: Movement threshold for detecting stop[0m [1;31mworking_distance: Working distance for pipettes[0m [1;34mdraw_contours: Draw contours?[0m [1;31mdraw_fitted_ellipses: Draw fitted ellipses?[0m

autofocus_size = 150

autofocus_sleep = 0.5

blur_size = 10

```
categories = [('Tracking', ['target_pixelperum', 'min_gradient', 'max_gradient',
'blur_size', 'minimum_contour', 'min_length', 'max_length', 'min_width',
'max_width', 'max_displacement']), ('Manipulation', ['working_distance',
'autofocus_size', 'autofocus_sleep']), ('Automation', ['stop_duration',
'stop_amplitude', 'minimum_stop_time']), ('Debugging', ['draw_contours',
'draw_fitted_ellipses'])]

draw_contours = False

draw_fitted_ellipses = False

max_displacement = 50

max_gradient = 98

max_length = 170

max_width = 60

min_gradient = 75

min_length = 65

min_width = 30

minimum_contour = 100

minimum_stop_time = 0

name = 'ParameciumDropletConfig'

stop_amplitude = 5

stop_duration = 50

target_pixelperum = 1

working_distance = 200
```

class holypipette.interface.paramecium_droplet.**ParameciumDropletInterface**(*pipette_interface*, *camera*)

Bases: *TaskInterface*

autofocus(*xy_position*)

Autofocus

Parameters
xy_position (*object, optional*)

autofocus_paramecium()

Autofocus on Paramecium

automatic_experiment()

Perform automatic experiment

detect_contact()

Detects contact of the pipette with water.

display_z_manipulator()

Display z position of manipulator relative to floor

focus()

Focus on tip

move_pipette_down()

Move pipette vertically to floor level

move_pipette_floor(*xy_position*)

Move pipettes to position at floor level

Parameters
xy_position (*object, optional*)

move_pipette_working_level(*xy_position*)

Move pipette down to position at working distance level

Parameters
xy_position (*object, optional*)

move_pipettes_paramecium()

Move pipettes to Paramecium

start_tracking(*xy_position*)

Start tracking paramecium at mouse position

Parameters
xy_position (*object, optional*)

toggle_following()

Toggle paramecium following

toggle_tracking()

Toggle paramecium tracking

track_paramecium(*frame*)

holypipette.interface.patch module

Control of automatic patch clamp algorithm

class holypipette.interface.patch.**AutoPatchInterface**(*amplifier*, *pressure*, *pipette_interface*)

Bases: *TaskInterface*

A class to run automatic patch-clamp

break_in()

Break into the cell

clean_pipette()

Clean the pipette (wash and rinse)

contact_detection()

Moving down the calibrated manipulator to detect the contact point with the coverslip

property current_autopatcher

patch_with_move(*position*)

Move to cell and patch it

Parameters
position (*object, optional*)

patch_without_move(*position=None*)

Patch cell at current position

Parameters
position (*object, optional*)

sequential_patching()

Sequential patching and cleaning for multiple cells

store_cleaning_position()

Store the position of the washing bath

store_rinsing_position()

Store the position of the rinsing bath

class holypipette.interface.patch.**PatchConfig**(*value_changed=None*, **args*, ***kwds*)

Bases: *Config*

params(Vramp_amplitude=NumberWithUnit, Vramp_duration=NumberWithUnit, cell_R_increase=Number, cell_distance=NumberWithUnit, gigaseal_R=NumberWithUnit, max_R=NumberWithUnit, max_R_increase=NumberWithUnit, max_cell_R=NumberWithUnit, max_distance=NumberWithUnit, min_R=NumberWithUnit, pressure_near=NumberWithUnit, pressure_ramp_duration=NumberWithUnit, pressure_ramp_increment=NumberWithUnit, pressure_ramp_max=NumberWithUnit, pressure_sealing=NumberWithUnit, seal_deadline=NumberWithUnit, seal_min_time=NumberWithUnit, zap=Boolean, name=String) [1;32mParameters of 'PatchConfig' ========== [0m [1;31mParameters changed from their default values are marked in red.[0m [1;36mSoft bound values are marked in cyan.[0m C/V= Constant/Variable, RO/RW = ReadOnly/ReadWrite, AN=Allow None

[1;34mName Value Type Bounds Mode [0m

pressure_near 20 NumberWithUnit (0, 100) V RW pressure_sealing -20 NumberWithUnit (-100, 0) V RW pressure_ramp_increment -25 NumberWithUnit (-100, 0) V RW pressure_ramp_max -300.0 NumberWithUnit (-1000, 0) V RW pressure_ramp_duration 1.15 NumberWithUnit (0, 10) V RW min_R 2000000.0 NumberWithUnit (0, 1000000000.0) V RW max_R 25000000.0 NumberWithUnit (0, 1000000000.0) V RW max_cell_R 300000000.0 NumberWithUnit (0, 1000000000.0) V RW cell_distance 10 NumberWithUnit (0, 100) V RW max_distance 20 NumberWithUnit (0, 100) V RW max_R_increase 1000000.0 NumberWithUnit (0, 100000000.0) V RW cell_R_increase 0.15 Number (0, 1) V RW gigaseal_R 1000000000.0 NumberWithUnit (100000000.0, 10000000000.0) V RW seal_min_time 15 NumberWithUnit (0, 60) V RW seal_deadline 90.0 NumberWithUnit (0, 300) V RW Vramp_duration 10.0 NumberWithUnit (0, 60) V RW Vramp_amplitude -0.07 NumberWithUnit (-0.2, 0) V RW zap False Boolean (0, 1) V RW

[1;32mParameter docstrings: ==========[0m

[1;34mpressure_near: Pressure during approach[0m [1;31mpressure_sealing: Pressure for sealing[0m [1;34mpressure_ramp_increment: Pressure ramp increment[0m [1;31mpressure_ramp_max: Pressure ramp maximum[0m [1;34mpressure_ramp_duration: Pressure ramp duration[0m [1;31mmin_R: Minimum normal resistance[0m [1;34mmax_R: Maximum normal resistance[0m [1;31mmax_cell_R: Maximum cell resistance[0m [1;34mcell_distance: Initial distance above target cell[0m [1;31mmax_distance: Maximum movement during approach[0m [1;34mmax_R_increase: Increase in resistance indicating obstruction[0m [1;31mcell_R_increase: Proportional increase in resistance indicating cell presence[0m [1;34mgigaseal_R: Gigaseal resistance[0m [1;31mseal_min_time: Minimum time for seal[0m [1;34mseal_deadline: Maximum time for seal formation[0m [1;31mVramp_duration: Voltage ramp duration[0m [1;34mVramp_amplitude: Voltage ramp amplitude[0m [1;31mzap: Zap the cell to break the seal[0m

```
Vramp_amplitude = -0.07

Vramp_duration = 10.0

categories = [('Approach', ['min_R', 'max_R', 'pressure_near', 'cell_distance',
'max_distance', 'cell_R_increase']), ('Sealing', ['pressure_sealing', 'gigaseal_R',
'Vramp_duration', 'Vramp_amplitude', 'seal_min_time', 'seal_deadline']),
('Break-in', ['zap', 'pressure_ramp_increment', 'pressure_ramp_max',
'pressure_ramp_duration', 'max_cell_R'])]

cell_R_increase = 0.15

cell_distance = 10

gigaseal_R = 1000000000.0

max_R = 25000000.0

max_R_increase = 1000000.0

max_cell_R = 300000000.0

max_distance = 20

min_R = 2000000.0

name = 'PatchConfig'

pressure_near = 20

pressure_ramp_duration = 1.15

pressure_ramp_increment = -25

pressure_ramp_max = -300.0

pressure_sealing = -20

seal_deadline = 90.0

seal_min_time = 15

zap = False
```

holypipette.interface.pipettes module

class holypipette.interface.pipettes.**PipetteInterface**(*stage*, *microscope*, *camera*, *units*, *config_filename=None*)

Bases: *TaskInterface*

Controller for the stage, the microscope, and several pipettes.

calibrate_manipulator()

Calibrate manipulator

calibrate_manipulator2()

Calibrate stage and manipulator (2nd Method)

`calibrate_stage()`

Calibrate stage only

`check_ranges()`

Check manipulator ranges

`connect(main_gui)`

Connect signals to slots in the main GUI. Will be called automatically during initialization of the GUI.

Parameters
main_gui (*CameraGui*) – The main GUI in control.

`go_to_floor()`

Go to the floor (cover slip)

`load_configuration()`

Load the calibration information

`manipulator_switched`

`measure_ranges()`

This is called every 500 ms when measuring ranges. It updates the min and max on each axis.

`move_microscope(distance)`

Move microscope by 10m

Parameters
distance (*object, optional*) – If no argument is given, 10 will be used as a default argument

`move_pipette(xy_position)`

Move pipette to position

Parameters
xy_position (*object, optional*)

`move_pipette_x(distance)`

Move pipette in x direction by 10m

Parameters
distance (*object, optional*) – If no argument is given, 10 will be used as a default argument

`move_pipette_y(distance)`

Move pipette in y direction by 10m

Parameters
distance (*object, optional*) – If no argument is given, 10 will be used as a default argument

`move_pipette_z(distance)`

Move pipette in z direction by 10m

Parameters
distance (*object, optional*) – If no argument is given, 10 will be used as a default argument

`move_stage(xy_position)`

Move stage to position

Parameters
xy_position (*object, optional*)

move_stage_horizontal(*distance*)

Move stage horizontally by 10m

Parameters
distance (*object, optional*) – If no argument is given, 10 will be used as a default argument

move_stage_vertical(*distance*)

Move stage vertically by 10m

Parameters
distance (*object, optional*) – If no argument is given, 10 will be used as a default argument

recalibrate_manipulator()

Recalibrate manipulator

recalibrate_manipulator_on_click(*xy_position*)

Recalibrate manipulator

Parameters
xy_position (*object, optional*)

reset_ranges()

Reset manipulator ranges

reset_timer()

Reset timer

save_configuration()

Save the calibration information

set_floor()

Set the position of the floor (cover slip)

switch_manipulator(*unit_number*)

Switch the currently active manipulator

Parameters
unit_number (*int*) – The number of the manipulator (using 1-based indexing, whereas the code internally uses 0-based indexing).

holypipette.utils package

Submodules

holypipette.utils.filelock module

A platform independent file lock that supports the with-statement.

class holypipette.utils.filelock.**BaseFileLock**(*lock_file*, *timeout=- 1*)

Bases: object

Implements the base class of a file lock.

acquire(*timeout=None*, *poll_intervall=0.05*)

Acquires the file lock or fails with a *Timeout* error.

```
# You can use this method in the context manager (recommended)
with lock.acquire():
    pass

# Or use an equivalent try-finally construct:
lock.acquire()
try:
    pass
finally:
    lock.release()
```

Parameters

- **timeout** (*float*) – The maximum time waited for the file lock. If `timeout < 0`, there is no timeout and this method will block until the lock could be acquired. If `timeout` is None, the default *timeout* is used.
- **poll_intervall** (*float*) – We check once in *poll_intervall* seconds if we can acquire the file lock.

Raises

Timeout – if the lock could not be acquired in *timeout* seconds.

Changed in version 2.0.0: This method returns now a *proxy* object instead of *self*, so that it can be used in a with statement without side effects.

property is_locked

True, if the object holds the file lock.

Changed in version 2.0.0: This was previously a method and is now a property.

property lock_file

The path to the lock file.

release(*force=False*)

Releases the file lock.

Please note, that the lock is only completly released, if the lock counter is 0.

Also note, that the lock file itself is not automatically deleted.

Parameters

force (*bool*) – If true, the lock counter is ignored and the lock is released in every case.

property timeout

You can set a default timeout for the filelock. It will be used as fallback value in the acquire method, if no timeout value (*None*) is given.

If you want to disable the timeout, set it to a negative value.

A timeout of 0 means, that there is exactly one attempt to acquire the file lock.

New in version 2.0.0.

holypipette.utils.filelock.**FileLock**

Alias for the lock, which should be used for the current platform. On Windows, this is an alias for *WindowsFileLock*, on Unix for *UnixFileLock* and otherwise for *SoftFileLock*.

class holypipette.utils.filelock.**SoftFileLock**(*lock_file*, *timeout=- 1*)

Bases: *BaseFileLock*

Simply watches the existence of the lock file.

exception holypipette.utils.filelock.**Timeout**(*lock_file*)

Bases: TimeoutError

Raised when the lock could not be acquired in *timeout* seconds.

lock_file

The path of the file lock.

class holypipette.utils.filelock.**UnixFileLock**(*lock_file*, *timeout=- 1*)

Bases: *BaseFileLock*

Uses the fcntl.flock() to hard lock the lock file on unix systems.

class holypipette.utils.filelock.**WindowsFileLock**(*lock_file*, *timeout=- 1*)

Bases: *BaseFileLock*

Uses the msvcrt.locking() function to hard lock the lock file on windows systems.

holypipette.vision package

Image processing algorithms

Submodules

holypipette.vision.crop module

Methods to crop images

holypipette.vision.crop.**crop_cardinal**(*image*, *direction*)

Returns a quadrant of the image corresponding to a cardinal point

Parameters

- **image** (*the image*)
- **direction** (*cardinal point as a string, in 'N', 'NW', 'S' etc*)

holypipette.vision.crop.**crop_center**(*image*, *ratio=32*)

Returns the center of the image.

Parameters

- **image** (*the image*)
- **ratio** (*size ratio of cropped image to original image*)

holypipette.vision.findpipette module

Methods to find the pipette in an image

holypipette.vision.findpipette.**pipette_cardinal**(*image*)

Determines the cardinal direction of the pipette (N, NW, S, etc) in the image.

holypipette.vision.findpipette.**pipette_cardinal2**(*image1*, *image2*)

holypipette.vision.findpipette.**up_direction**(*pipette_position*, *positive_move*)

Determines the direction (+1 or -1) of the pipette going up.

Parameters

- **pipette_position** (*cardinal position of the pipette*)
- **positive_move** (*vector of image movement for a positive displacement along the axis*)

holypipette.vision.paramecium_tracking module

These are functions to locate paramecium in an image

class holypipette.vision.paramecium_tracking.**ParameciumTracker**(*config=None*, *history_size=100*)

Bases: object

clear()

has_stopped()

locate(*frame*, *pixel_per_um*)

Locate paramecium in an image.

Parameters

- **frame** – the image
- **pixel_per_um** (*float*) – number of pixels per µm

Returns
x, y, MA, ma, angle

Return type
Position and size of fitted ellipse

median_position(*look_back=None*)

holypipette.vision.paramecium_tracking.**where_is_droplet**(*frame*, *pixel_per_um=5.0*, *ratio=None*, *xc=None*, *yc=None*)

Locate a droplet in an image.

Parameters

- **frame** (*the image*)
- **pixel_per_um** (*number of pixels per um*)
- **ratio** (*decimating ratio (to make the image smaller)*)
- **xc, yc** (*coordinate of a point inside the droplet*)

Returns
x, y, r

Return type
position and radius on screen

holypipette.vision.paramecium_tracking.**where_is_paramecium2**(*frame*, *pixel_per_um=5.0*, *return_angle=False*, *previous_x=None*, *previous_y=None*, *ratio=None*, *background=None*, *debug=False*, *max_dist=1000000.0*)

holypipette.vision.phase_cross_correlation module

Copied from scikit-image.registration 0.18.0. Masking removed.

holypipette.vision.phase_cross_correlation.**phase_cross_correlation**(*reference_image*, *moving_image*, *upsample_factor=1*, *space='real'*, *return_error=True*, *overlap_ratio=0.3*)

Efficient subpixel image translation registration by cross-correlation. This code gives the same precision as the FFT upsampled cross-correlation in a fraction of the computation time and with reduced memory requirements. It obtains an initial estimate of the cross-correlation peak by an FFT and then refines the shift estimation by upsampling the DFT only in a small neighborhood of that estimate by means of a matrix-multiply DFT. :Parameters: * **reference_image** (*array*) – Reference image.

- **moving_image** (*array*) – Image to register. Must be same dimensionality as `reference_image`.
- **upsample_factor** (*int, optional*) – Upsampling factor. Images will be registered to within `1 / upsample_factor` of a pixel. For example `upsample_factor == 20` means the images will be registered within 1/20th of a pixel. Default is 1 (no upsampling). Not used if any of `reference_mask` or `moving_mask` is not None.
- **space** (*string, one of "real" or "fourier", optional*) – Defines how the algorithm interprets input data. "real" means data will be FFT'd to compute the correlation, while "fourier" data will bypass FFT of input data. Case insensitive. Not used if any of `reference_mask` or `moving_mask` is not None.
- **return_error** (*bool, optional*) – Returns error and phase difference if on, otherwise only shifts are returned. Has noeffect if any of `reference_mask` or `moving_mask` is not None. In this case only shifts is returned.
- **overlap_ratio** (*float, optional*) – Minimum allowed overlap ratio between images. The correlation for translations corresponding with an overlap ratio lower than this threshold will be ignored. A lower `overlap_ratio` leads to smaller maximum translation, while a higher `overlap_ratio` leads to greater robustness against spurious matches due to small overlap between masked images. Used only if one of `reference_mask` or `moving_mask` is None.

Returns

- **shifts** (*ndarray*) – Shift vector (in pixels) required to register `moving_image` with `reference_image`. Axis ordering is consistent with numpy (e.g. Z, Y, X)
- **error** (*float*) – Translation invariant normalized RMS error between `reference_image` and `moving_image`.
- **phasediff** (*float*) – Global phase difference between the two images (should be zero if images are non-negative).

References

holypipette.vision.templatematching module

Search a template image in an other image Not scale nor rotation invariant

Uses OpenCV. Alternatively, one might use skimage.feature.match_template

exception holypipette.vision.templatematching.**MatchingError**(*value*)

Bases: Exception

holypipette.vision.templatematching.**templatematching**(*img*, *template*, *threshold=0*)

Search a template image in an other image Not scale nor rotation invariant.

Parameters

- **img** (*image to look in*)
- **template** (*image to look for*)
- **threshold** (*throw an error if match value is below threshold*)

Returns

- **x** (*x coordinate of the template in the image*)
- **y** (*y coordinate*)
- **maxval** (*maximum value corresponding to the best matching ratio*)

2.1.2 Submodules

holypipette.config module

Support for configuration objects (based on the param package)

class holypipette.config.**Config**(*value_changed=None*, **args*, ***kwds*)

Bases: Parameterized

params(name=String) [1;32mParameters of 'Config' ========== [0m Object has no parameters.

from_dict(*config_dict*)

from_file(*filename*)

name = 'Config'

to_dict()

to_file(*filename*)

class holypipette.config.**NumberWithUnit**(*default*, *unit*, *magnitude=1.0*, **args*, ***kwds*)

Bases: Number

A numeric Dynamic Parameter, with a default value and optional bounds.

There are two types of bounds: `bounds` and `softbounds`. `bounds` are hard bounds: the parameter must have a value within the specified range. The default bounds are (None,None), meaning there are actually no hard bounds. One or both bounds can be set by specifying a value (e.g. bounds=(None,10) means there is no lower

bound, and an upper bound of 10). Bounds are inclusive by default, but exclusivity can be specified for each bound by setting inclusive_bounds (e.g. inclusive_bounds=(True,False) specifies an exclusive upper bound).

Number is also a type of Dynamic parameter, so its value can be set to a callable to get a dynamically generated number (see Dynamic).

When not being dynamically generated, bounds are checked when a Number is created or set. Using a default value outside the hard bounds, or one that is not numeric, results in an exception. When being dynamically generated, bounds are checked when the value of a Number is requested. A generated value that is not numeric, or is outside the hard bounds, results in an exception.

As a special case, if allow_None=True (which is true by default if the parameter has a default of None when declared) then a value of None is also allowed.

A separate function set_in_bounds() is provided that will silently crop the given value into the legal range, for use in, for instance, a GUI.

softbounds are present to indicate the typical range of the parameter, but are not enforced. Setting the soft bounds allows, for instance, a GUI to know what values to display on sliders for the Number.

Example of creating a Number:

```
AB = Number(default=0.5, bounds=(None,10), softbounds=(0,1), doc='Distance from A␣
↪to B.')
```

magnitude

unit

holypipette.log_utils module

class holypipette.log_utils.**LoggingObject**

Bases: object

debug(*message*, **args*, ***kwds*)

error(*message*, **args*, ***kwds*)

exception(*message*, **args*, ***kwds*)

info(*message*, **args*, ***kwds*)

property logger

warn(*message*, **args*, ***kwds*)

holypipette.log_utils.**console_logger**()

PYTHON MODULE INDEX

h

INDEX

A

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z