



itom Documentation

Release 8227a0e

Institut fuer Technische Optik (ITO), University Stuttgart

July 22, 2015

1	Changelog	1
1.1	itom	1
1.2	Plugins	9
1.3	Designer Plugins	12
2	Introduction	17
2.1	About us	18
2.2	Licensing	18
3	Installation	21
3.1	Minimum system requirements	21
3.2	Installation from setup	21
3.3	Build from Sources	34
3.4	Plugins, Designer-Plugins	52
3.5	All-In-One development setup	52
3.6	Get this help	56
4	Getting Started	61
4.1	Quick Start	61
4.2	Further Information	72
4.3	Tutorials	72
5	The itom User Interface	73
5.1	Main Window	73
5.2	Script-Editor	83
5.3	Property Dialog	88
5.4	Python Package Manager	93
6	Plots and Figures	99
6.1	Quick tutorial to plots and figures	99
6.2	Figure Management	104
6.3	line plots (1D)	106
6.4	2D image plots	109
6.5	isometric Plot	113
6.6	Primitives - Marking and Measuring	114
6.7	Matplotlib	119
7	Extending the user interface of itom	121
7.1	Customize the menu and toolbars of itom	121
7.2	Show messages, input boxes and default dialogs	123
7.3	Creating advanced dialogs and windows	128
7.4	Custom Designer Widgets	140
7.5	Implement a more complex GUI in a plugin (C/C++)	144
8	Plugins	145
8.1	Basic concept	145

8.2	How to start and use a plugin	145
8.3	Development under C++	147
9	Python scripting language	293
9.1	Introduction	293
9.2	Python module itom	332
9.3	Further python packages	338
9.4	Tutorials, documentations about Python 3	343
10	itom Script Reference	345
10.1	itom methods	345
10.2	dataIO	353
10.3	actuator	360
10.4	Algorithms, Widgets and Filters	366
10.5	figure	367
10.6	plotItem	369
10.7	ui-elements (ui, uiItem)	369
10.8	dataObject	379
10.9	point	394
10.10	pointCloud	396
10.11	polygonMesh	400
10.12	region	401
10.13	rgba	401
10.14	autoInterval	402
10.15	timer	402
10.16	font	403
11	Miscellaneous	405
11.1	How to use the help	405
11.2	Units and Conventions	405
12	Demo scripts	407
12.1	itom Basics	407
12.2	Plugins	407
12.3	Algorithm / Filter	408
12.4	ui	408
13	Python tutorials	409
13.1	determine lateral image shift and show images by using itom figure plots	409
14	Indices and tables	413
	Python Module Index	415

CHANGELOG

This changelog only contains the most important changes taken from the commit-messages of the git.

1.1 itom

Version 2.0.0 (2015-07-20)

(more than 290 commits in itom repository)

- easier compilation under Windows 8
- welcome message removed since it bothers upon regular usage.
- modified icons and splash screen in preparation for upcoming major version 2.0.
- demo about saving and loading data objects added
- default plugins for category *DObjLiveImage* changed to *itom2dqwtplot* and *PerspectivePlot* to *twipOGLFigure*
- script added in plugin help section to automatically parse the parameter section in a plugin's documentation file from a running instance of the plugin
- some improvements when pressing key up or down in console while also pressing the ctrl or shift buttons
- plots (e.g. *itom1dqwtplot* and *itom2dqwtplot*) can now display axes labels in various ways. Enum *UnitLabelStyle* added.
- It is now possible to a translation file (qm) to a python-loaded ui file
- added linguist application to setup to create user defined translations for user-defined GUIs.
- addin interface incremented to 2.0.0: branch embedded line plot merged, read-write-lock of data object removed, deprecated methods removed, cleanup in *AddInBase*, *AddInActuator* and *AddInDataIO* (things from SDK-change issue on bitbucket)
- embedded line plot added to allow redirecting a line cut or z-stack in the designer plugin *itom2dqwtplot* to an existing *itom1dqwtplot* that is already contained in an user-defined GUI (demo available)
- improvements in plugin parameter validator: *rectMeta* information is now better checked with more precise error messages
- setup comes now with Numpy 1.9.2 (MKL-version)
- algorithm plugin auto documentation improved to also show parameters and descriptions of filters
- *itom.font* for wrapping font information. This allows changing *axisFont*, *labelFont*... properties of plots
- fix to convert between QColor-property and Python (itom.rgba, hex-color, string-color-name) in both directions
- improved documentation style sheet for Qt5

- `itom.actuator` and `itom.dataIO` can now be observed using weak references (python module `weakref`)
- deprecated class `itom.npDataObject` removed.
- `demoFitData.py` added to demo files to show how to fit 2D polygons to data.
- `itom.timer` now accepts an interval of 0ms. The interval must now always be an integer. Further argument `singleShot` added to constructor in order to allow a single shot timer that only fires once after a start.
- If “find word” widget in script editor is opened and user presses Esc key, the widget will be closed and the focus set to the parent script again.
- fix such that running or debugging script snippets in the command line always print the result of every evaluated command (even in multiline commands).
- bugfixes concerning changed default encoding between Qt4/Qt5. `itom`’s default encoding is latin1 (like Qt4). In Qt5 it changed to utf-8.
- `children()` added to get a dictionary with all children of this `uiItem` (widget), optional: recursive scan of sub-children, too.
- *insert codec string* feature added to script editor to insert the `# coding=...` line in any script.
- help dock widget can now also be undocked as main window (like scripts or plots)
- let user choose if unsaved files should always/never be saved before script execution or debugging (re-vertable via property dialog)
- introduction of Python Package Manager (via module `pip`). This dialog allows updating and installing python packages and its dependencies from `pypi.python.org` or wheel files.
- `fromXYZRGBA()` and `fromOrganizedCloud()` added.
- demo about statusbar in main windows added
- fix in `DataObject::mul` and `DataObject::div`: axis tags and tag map from first operand are copied to resulting object.
- new api methods `apiSendParamToPyWorkspace` and `apiSendParamsToPyWorkspace` added.
- removal of unused `DataObject::lockRead()`, `DataObject::unlock()`, `DataObject::lockWrite()` methods.
- support for Mac OSX, 64bit. The build process is mainly supported by the package *brew*.
- virtual slots `currentRow` and `currentColumn` added to `QTableWidget` for accessing these slots via Python.
- negative indices of `dataObject` indexing is now allowed and wraps around: `obj[-1,-1]` accesses the last element.
- change in `ito::dObjHelper::squeezeConvertCheck2DDataObject`: parameter `convertToType`: -1 means no conversion will be done, this was 0 before. However 0 collides with `int8`.
- `phyToPix()` and `pixToPhys()` added
- `info()` can now print more properties, slots and signals if called with parameter 1 or 2.
- unifications of OS-dependent macros. For `itom`, plugins and `designerPlugins` the following pre-compiler directives are set: `WIN32`, `_WIN32` for all Win-Systems; additionally `WIN64`, `_WIN64` for x64 build; `linux`, `Linux` for Linux(Unix) based systems, `__APPLE__` for Apple systems.
- added `gitignore` with respect to `linux`, `windows`, `osx`, `c++`, `xcode`, `visualstudio`, `tortoisegit`, `cmake`, `python`, `ipythonnotebook`, `qt`
- compare operators with real, scalar operand inserted for data object (support in C++ and Python `itom.dataObject`).
- mapping set of data object can now get a mask (e.g. `itom.dataObject a; a[b > 0] = 2`).
- `setTo`-method and `at`-method with mask parameter added to `DataObject`. It is then possible to set a scalar value to masked values or to return a new data object that only contains masked values in a `1xM` object.

- bugfix: designer.exe could not be started in setup environment (Windows). This is fixed now.
- `createMask()` can get a bounding rectangle such that the returned mask data object can be adjusted concerning its size and offset.
- many bugfixes

Version 1.4.0 (2015-02-17)

(more than 200 commits in itom repository)

- improved getting started documentation
- CMake adapted for detection Visual Studio versions > 2010
- `dockWidgetArea` added as optional parameter for `itom.ui`-class such that windows of type `ui.TYPEDOCKWIDGET` can be placed at different locations.
- `find_package(OpenMP)` added to `ItomBuildMacros` for all compilers, since Visual Studio Express does not provide this tool.
- improved drag&drop from variable names from workspace to console
- redesigned user management dialogs
- python access key to variables in workspace widget can now be inserted into console or scripts via drag&drop, detailed variables window also contain the right access key for dicts, list, tuples, attributes...
- `itom.plotItem.PrimitiveEllipse`, `PrimitiveLine...` added as constants to the class `plotItem` of itom module in Python.
- double click on Python error message (filename, line-number) in console opens the script at the indicated line
- increased buffer size of data object protocol strings since `sprintf` crashes upon buffer overflow (even `sprintf_s`, debug assertion)
- Improved `itom.plotHelp()`-command to get a list of all plot widget
- new demo scripts added (see demo subfolder)
- `pointCloud.copy()` for deep copies added
- Make Python loops interruptible from GUI
- Added functions to edit, add and read out color maps bases on rgba-values
- `itom.addMenu()` / `itom.removeMenu()` / `itom.addButton()` / `itom.removeButton()`: the methods to add a button or menu element always return an unique handle to the recently added item. The methods to remove them can be called with this handle to exactly delete the element that has been added (does not remove an element with the same name that replaced the previously added one). With this change, a bug has been fixed: In some cases methods or functions connected to menu items have not been released when deleting the parent menu element.
- `observeInvocation` in `AbstractAddInConfigDialog` and `abstractAddInDockWidget` now use the `waitAndProcessEvent` method of `ItomSharedSemaphore` instead of a “self-programmed” version doing the same.
- `itom.addButton()` returns handle to button, this handle can be used to call `:py:meth:itom.removeButton'(handle)` to exactly delete this button. The existing method `:py:meth:itom.removeButton'(toolbarName, buttonName)` still exists as other possibility
- `AddInManager::initAddIn`: let this method also processEvents when it waits for the plugin's init method to be finished. Then, the plugin can directly invoke slots from GUI-related elements of the plugin within its init method. * Bugfix in `dataObject.toGray()` with default parameter (type)
- `QPropertyEditor` can now handle `QStringList` properties (with list editor dialog for editing the `QStringList`)
- methods `removeItem`, `setItemData` and `insertItem` of `QComboBox` are now callable from Python (via `.call(...)` command)
- Qt5 fixes and adaptations (encodings, qmessagehandler, linkage against shell32 for Qt5.4 and Windows, ...)

- ‘log’ can be passed as argument to the executable to redirect debug outputs to itomlog.txt.
- always set Environmental Variable MPLCONFIGDIR to itom-packages/mpl_itom such that an individual matplotlibrc config file can be placed there. You can set the backend within this config file to module://mpl_itom.backend_itomagg such that matplotlib always renders to itom windows if used within itom.
- All-In-One build environment for Qt5, Python 3.4, VS2010, 32 or 64bit, PCL 1.8.0, VTK 6.1 created. Documentation about all-in-one build environment added.
- Added more flexible thread-safe control classes for all types of dataIO (grabber, ADDA, rawIO) and actuators (using inheritance).
- added multi-parameter slot getParamVector to ito::AddInBase (similar to setParamVector).
- class CameraThreadCtrl inserted in helperGrabber.h (part of SDK; replacement for threadCamera). The new class guards the camera by temporarily incrementing its reference counter, checks the isAlive() flag when waiting for semaphores and directly frees the semaphore after usage. A camera pointer can directly be passed to this class.
- some checks inserted to avoid crashes at startup if Python could not be found.
- theme for html documentation adapted
- replaced c-cast in REMOVE_PLUGININSTANCE by qobject_cast (due to some rare problems deleting plugins)
- documentation: sphinx extension Breathe updated to version 3.1
- improved parameter validation for floating point parameters (consider epsilon value)
- typedef ParamMap and ParamMapIterator as simplification for QMap<QString,ito::Param> ... inserted
- better detection of OpenCV under linux
- fix in backend of matplotlib (timer from threading package to redraw canvas for instance after a zoom or home operation did not work. Replaced by itom.timer)
- range: (min,max,step) and value: (min,max,step) also added for rangeWidget of itomWidgets project. rangeSlider and rangeWidget can now also be parametrized passing an IntervalMeta or RangeMeta object.
- RangeSlider widgets now have the properties minimumRange, maximumRange, stepSizeRange, stepSizePosition
- some new widgets added to itomWidgets project
- IntervalMeta, RangeMeta, DoubleIntervalMeta, CharArrayMeta, IntArrayMeta, DoubleArrayMeta, RectMeta added as inherited classes from paramMeta. AddInInterface incremented to 1.3.1.
- bugfix in dataObject when creating a huge (1000x500x500, complex128) object (continuous mode).
- method uiItem.exists added in order to check if the widget, wrapped by uiItem, still exists.
- bugfix in squeezeConvertCheck2DDDataObject if numberOfAllowedTypes = 0 (all types allowed)
- Python script navigator added to script editor window (dropdown boxes for classes and methods)
- implicit cast of ito::Rgb32 value to setVal<int> of ito::Param of type integer.
- improved integration of class itom.autoInterval and the corresponding c++ class ito::AutoInterval. Conversion to other data types added.
- documentation about programming of AD-converter plugins (dataIO, type ADDA) added
- several bugfixes in breakPointModel (case-insensitivity of filenames, better error message if breakpoint is in empty line, ...)
- many other bugs fixed
- improvements in documentation

Version 1.3.0 (2014-10-07)

(more than 150 commits in itom repository)

- fixes big bug in assignment operator of dataObjects if d2=d1 and d1 shares its data with an external object (owndata=0). In the fixed version, d2 also has the owndata flag set to 0 (before 1 in any cases!)
- replace dialog for replacing text within a selection fixed
- API function `apiValidateAndCastParam` added. This is an enhanced version of `apiValidateParam` used in `setParam` of plugins. The enhanced version is able to modify possible double parameters in order to fit to possibly given step sizes.
- support of PyMatlab integrated into CMake system (check `BUILD_WITH_PYMATLAB` to enable the python module 'matlab' and indicate the include directory and some libraries of Matlab)
- Add twipOGL-Plugin (from twip Optical Solutions GmbH) to `plotToolBar` if present
- dataObject: `axisUnits`, `axisDescriptions`, `valueUnit` and `valueDescription` can now handle encoded strings (different than UTF8)
- `QDate`, `QDateTime` and `QTime` are now marshalled to python `datetime.date`, `datetime.datetime` and `datetime.time` (useful for `QDateTimeEdit` or `QDateEdit` widgets)
- text from console widget is never deleted when drag&dropping it to any script widget
- state of python reloader is stored and restored using the settings
- warning is displayed if figure class could not be found and default fallback is applied
- `itom.plotLoaded()` and `plotHelp()` added
- Improved protocol functionality for dataObject related python functions
- Fixed missing copy of `rotationMatrix` metadata for `squeeze()` function in `dataobj.cpp`
- Added `copyAxisTagsTo` and `copyTagMapTo` to `ito::absFunc`, `ito::realFunc`, `ito::imagFunc`, `ito::argFunc` to keep `dataTags`
- example about modal property dialogs added to demo folder
- `QScintilla` version string added to version helper and about dialog
- `itom.clc()` added to clear the command line from a script (useful for overfull command line output)
- `AutoInterval` class published in common-SDK. This class can be used as datatype for an min/max-interval (floats) with an additional auto-flag.
- public methods **`selectedRows`**, **`selectedTexts`** and **`selectRows`** of `QListWidget` can now be called via `itom.uiItem.call()`
- `itom.dataObject` operator `/` and `/=` for scalar operand implemented (via inverse multiplication operator) fixed casting issue in multiplication operator for double scalar (multiply with double precision -> then cast)
- configured `QPropertyEditor` as shared library (dll) instead of static library. This was the last library that was not shared yet.
- reduced size of error messages during live image grab
- fix when reloading module, that contains syntax or other errors: these errors are displayed for better handling
- If an instance of `QtDesigner` is opened and then an ui-file is loaded, this file was opened in a new instance of `QtDesigner`. This is fixed now.
- some crashes when not starting with full user rights are fixed
- Added demofile for data fitting
- if last tab in script window is closed, the entire window is closed as well
- types `ito::DataObject`, `QMap<QString, ito::Param>...` are no registered for signal/slot in `addInManager` such that this needs not to be done in plugins any more.
- class **`enum`** in `itomPackages` renamed to **`itomEnum`** since Python 3.3 introduces its own class **`enum`**.
- check for AMD-CPU's and adjust environment variable for these processors in order to avoid a `KMP_AFFINITY` error using Matplotlib (and OpenMP).

- enhanced output for class function `itom.ui.info()`.
- optional properties argument added to commands `itom.plot()` and `itom.liveImage()`. Pass a dictionary with properties that are directly applied to the plot widget at startup.
- recently opened file menu added to itom main window and script windows
- improved loaded plugins dialog
- some fixes in data object: fixed constructor for existing `cv::Mat` and type `RGBA32`, fixed bug in assignment operator for rhs-dataObjects that do not own their data block.
- property dialog documented
- improved python module auto reloader integrated into itom (based on iPython implementation). This reloader can be enabled by the menu script `>> reload modules`.
- some examples from the matplotlib gallery added to the demo scripts
- bugfix when changing the visibility of undocked plots
- the designer plugin *matplotlib* has now the same behaviour than other plot widgets (can be docked into main window, property toolbox available...)
- some improvements with tooltip texts displaying optional python syntax bugs (python module frosted required)
- unified signatures passed to `Q_ARG`-macro of Qt.
- the search bar is not hidden again if `Ctrl-F` is pressed another time.
- detailed descriptions of plugins are now also displayed in help toolbox
- improvements to reject the insertion of breakpoints in commented or empty lines
- improved breakpoint toolbox that allows en-/disabling single or all breakpoints, deleting breakpoints... Breakpoints are reloaded at new startup.
- unused or duplicated code removed and cleaned
- project *itomWidgets* synchronized to changes in mother project *commonCTK*
- german translations improved
- itom and the plugins now support a build with Qt5. The setup is still compiled with Qt4.8.
- support for CMake 3.0 added

Version 1.2.0 (2014-05-18)

(more than 300 commits in itom repository)

- `itom.dataObject.toGray()` added in order to return a grayscale dataObject from type `rgba32`.
- Drag&drop python files to command line in order to open them
- `itom.actuator.setInterrupt()` added in order to interrupt the current movement of an actuator (async-mode only, general interrupt must be implemented by plugin)
- Window bugfix (improper appearance) in Qt5 (windows) as well as some linux distributions (Qt4/Qt5)
- Breakpoint toolbox improved.
- Open scripts and current set of breakpoints is saved at shutdown and restored at new startup.
- Python, Numpy, Scipy, Matplotlib and itom script references can be automatically downloaded and updated from an itom online repository. These references can then be shown in the itom help toolbox.
- CMake: Improvements in `find_package(ITOM_SDK...)` COMPONENTS `dataObject`, `pointCloud`, `itomWidgets`, `itomCommonLib` and `itomCommonQtLib` can be chosen. Since `dataObject` and `pointCloud` require the core components of `OpenCV` and `PCL` respectively, these depending libraries are detected as well and automatically added to `ITOM_SDK_LIBRARIES` and `ITOM_SDK_INCLUDE_DIRS`.

- `itom.npDataObject` removed (reasons see: <http://itom.58578.x6.nabble.com/itom-discussions-deprecated-class-itom-npDataObject-td3.html>)
- Designer plugins can now display point clouds and polygon meshes (`itomIsoGLFigurePlugin` displays point-Clouds)
- improved codec handling. Default codec is latin1 (ISO 8859-1) now, Python's codec is synchronized to Qt codec.
- Modified / new templates added for creating a grabber, actuator and algorithm plugin.
- `ito::DataObject` registered as meta type in itom. Plugins do not need to do this any more.
- Styles in property editor can be reset to default, size of all styles can be globally increased or decreased
- C-API: new base classes for plugin configuration dialogs and dock widgets added to SDK: `ito::AbstractAddInDockWidget` and `ito::AbstractAddInConfigDialog`
- modified and improved user documentation
- itom extension added for sphinx for semi-automatic plugin documentation
- More CMake macros added for easier implementation of plugins: `FIND_PACKAGE_QT`, `PLUGIN_TRANSLATION`, `PLUGIN_DOCUMENTATION`
- Python commands `itom.pluginHelp()` and `itom.filterHelp()` either return dict with information or print information (depending on arguments)
- Plugin can provide a rst-based documentation file. The documentation of all available plugins is then integrated in the itom help system.
- Python syntax check in script editors introduced. Install the python package **frosted** in order to obtain the syntax check (can be disabled in property dialog)
- meta information `stepSize` introduced for `ito::IntMeta` and `ito::DoubleMeta`
- glew is not necessary if building against Qt5 with OpenGL.
- New shared library **itomWidgets** added to SDK. itom, user defined GUIs and plugins have now access to further widgets. These widgets are mainly taken from the CTK project (the common toolkit).
- Python class **enum** in itom-packages renamed to **itomEnum** since a class enum (with similar functionality is introduced with Python >= 3.4)
- All `signaling_NaNs` replace by `quiet_NaNs` (`signaling_NaNs` raise C exception under certain build settings)
- Unittests for `ito::ByteArray`, `ito::RetVal` and `ito::Param` added
- User management partially available via Python.
- itom can now be build and run using Qt4 or Qt5. Usually the Qt-version installed on the computer is automatically detected, if both versions are available use the CMake variable `BUILD_QTVERSION` and set it to Qt4 or Qt5 for manual choice.
- Help toolbox also shows information about all loaded hardware plugins (actuator, dataIO)
- Help toolbox in itom shows information about filters and widgets provided by algorithm plugins. Exemplary code snippet for all filters added.
- itom can be build without PCL support, hence the point cloud library is not required. The library *pointCloud* is not available then (see `BUILD_WITH_PCL` in CMake)
- C-API: New shared libraries created: `itomCommon` (`RetVal`, `Param`,...), `itomCommonQt` (`ItemSharedSemaphore`, `AddInInterface`,...), `dataObject`, `pointCloud`, `itomWidgets`. Plugins link againsts these shared libraries. This allows an improved bugfix in these components. Additionally, many changes in these libraries don't require a new version number for the plugin interface (unless the binary compatibility is not destroyed).
- C-API: error message in `ito::RetVal`, name and info string of `ito::Param` are now stored as `ito::ByteArray` class that is based on a shared memory concept. Less memory copy operations are required.

- crash fixed if itom is closed with user defined buttons and menus
- fixes if some components are disabled by user management
- C-API: DesignerPlugins can now be used within other plugins (e.g. widgets in algorithm plugins)
- many bugfixes

Version 1.1.0 (2014-01-27)

- help dock widget to show a searchable, online help for the script reference of several python packages, the itom module as well as a detailed overview of algorithms and widgets contained in plugins
- revised documentation and python docstrings
- optimization due to tool *cppCheck*
- method *apiGetParam* added to api-functions
- timeouts changeable by ini-file
- size, position... of native dock widgets and toolbars is saved and reloaded at restart
- further demos added
- property editor for plots added
- compilation without PointCloudLibrary possible (CMake setting)
- easier compilation for linux
- 2DQwtPlot enhanced within a code sprint to support geometric primitives that can be painted onto the plotted image. The parameters of the geometries can then be obtained via python and for instance be evaluated in the designer widget 'evaluateGeometries'. Demo script added for demonstrating this functionality (*uiMeasureToolMain demo*)
- many methods of dataObject now have int-parameters or return values instead of size_t -> better compatibility with respect to OpenCV
- In *itom.uiItem* it is now possible to also assign a string (or an integer) to enumeration based properties of widgets.
- *itom.openScript()* enhanced to also open the script where any object (class, method...) is defined in (if it exists)
- *itom.dataObject* have the attributes 'ndim' and 'shape' for a better compliance to 'numpy.ndarray'
- color type 'rgba32' added for data objects. See also *itom.rgba*. The color class in C++ is contained in color.h. 2dgraphicview plot also supports color cameras. OpenCVGrabber can grab in color as well. Unittest of data object adapted to this.
- better exception handling of any exception occurring in plugin algorithms.
- type 'ui.DOCKWIDGET' now possible for *itom.ui* in order to dock any user defined window into the main window
- drag&drop from last-command dock widget to console or script window
- modified python debugger
- added *itom.compressData()* and *itom.uncompressData()* in *itom*
- normalize method for data objects added
- many bugfixes

Version 1.0.14 (2014-09-02)

there is no continuous changelog for these version

1.2 Plugins

Version 2.0.0 (2015-07-20)

(more than 250 commits in plugins repository)

- **Ximea**: Renewed plugin tested with several xiQ cameras (MQ013, MQ042). More parameters like full-featured triggering, frame_burst, fixed and auto framerate... The parameter trigger_mode2 was renamed to trigger_selector (what it really is). Updated to XIMEA API 4.0.0.5. A software based shading correction was added, too. Meta information added to each captured frame.
- **PointGrey FlyCapture (PGRFlyCapture)**: improved documentation; if a slow data connection (e.g. USB2) is used, some data packages might be lost, therefore we retry to fetch the image if it could not be obtained the first time. All parameters renamed to underscore-separated version. 'roi' parameter added instead of x0,x1,y0,y1. Further parameters *trigger_polarity* and *packetize* added. Meta information added to each captured frame.
- **LibUSB**: Many improvements: Information about all connected devices and their endpoints (if device is readable by libusb) can be printed, different endpoints for reading and writing are adjustable. Improved documentation.
- **PCOSensicam** plugin added: This plugin supports image acquisition of the Sensicam camera series from PCO.
- **PCOPixelFly**: bugfixes and modified dock widget as well as configuration dialog. Gain is a two-state variable only (0: default mode, 1: higher infrared sensitivity)
- **USBMotion3XIII**: some smaller bugfixes in this motion controller plugin
- motor controller plugin **Standa 8SMC4USB** added.
- documentation of **AVTVimba** improved
- version 1.0 of **DummyGrabber** released: many bugfixes, binning, roi change, integration time, frame time, gain and offset are now working and influence the resulting image.
- plugin **ThorlabsCCS** (spectrometer from Thorlabs): 'roi' parameter added.
- plugin **AvantesAvaSpec** added to run spectrometers from Avantes only using the **LibUSB** plugin without need of further SDKs or drivers from Avantes.
- improvements in plugin **OpenCVGrabber**: native setting dialog for DirectShow based cameras can be opened in configuration dialog. This dialog provides more settings. Parameter *roi* added
- bugfix: avoid crash at shutdown of **glDisplay** in Debug mode, Qt5
- **libmodbus**: modbus-function read/write coils implemented for uint8 - data objects
- plugin **CommonVisionBlox** added to control cameras that are accessed via CommonVisionBlox from Stemmer (GenICam based).

68fb0e0 roi of OpenCVGrabber is now adjustable * plugin **PCOCamera** for the general camera SDK from PCO: parameter *roi* added * plugin **OpenCVFilters**: some bugfixes, filters *findHomography* added, fixes in **cvFlannBasedMatcher** * plugin **FittingFilters** is now compiled with *Lapacke*, filters *polyfitWeighted2DSinglePoints* and *polyval2DSinglePoints* added to FittingFilters to also fit arbitrary sets of x,y,z points. * bugfix in centroid1D (**dataobjectarithmetic**) if input object has scale!=1 or offset!=0. Version: 0.0.2 * modified version 1.0 of **DummyMotor** released (config dialog based on AbstractAddInConfigDialog) * fix in remote/local detection of **PIPiezoCtrl** * fix in load and save x3p: scale values are fixed. When loading, one can indicate the desired x,y and value units (m, cm, mm, Åµm, nm). When saving in x3p format, the axis and value units are parsed such that all units are scaled to meter, which is default in x3p. default units m. This is more robust since it is also the default of x3p. Other default units might lead to value multiplications that can cause overflows for certain data types; the user should consider this. z-axis must always be absolute, but can contain a offset (translation vector, z-component). z-scaling is always multiplied to values. simplifications in loading data: x3p data types are directly mapped to one dataObject type (no complex switch cases necessary). * filters *calcRadialMean* and *spikeMeanFilter* added to **BasicFilters** * plugin **NI-DAQmx**: improvements in NI-DAQmx: device as optional parameter for initialization inserted in order to indicate the name of the device (e.g. Dev1).

Tasks can now be restarted. * Plugin **PclTools**: filter **pclDistanceToModelDObj** and **saveVTKImageData** added (allows displaying volume plots e.g. in ParaView) * Plugin **DataObjectIO**: improvements and enhanced documentation in all save and load filters for image formats (png, tif, jpg, pgm, ...). These filter can now load color and monochrome image formats including possible alpha channels. Many filters improved, fixed and tested. load-NistSDF now supports ascii NIST, ISO (ISO25178-71) or the original BCR standard (used by Zygo NewView as export format) * many other bugfixes, especially concerning Qt5

Version 1.4.0 (2015-02-17)

(more than 170 commits in plugins repository)

- vertex and fragment shader error in gl based plugins: Since NVIDIA 347.xx, no character (including space, n...) must be before #version directive. Else shaders may not be compiled (error C0204, version directive must be first statement)
- linux support for ttyS, ttyUSB and ttyACM in plugin **serialIO** added, selection via portnumber S=0-999 USB=1000-1999 ACM=2000-2999
- some fixes in **FireGrabber**
- fix a crash in **PCOPixelFly** if camera is not correctly connected with the board
- fix in **OpenCVGrabber** when using the EasyCAP or similar analog-digital video converters (USB2)
- filters *cvResize*, *cvSplitChannels*, *cvMergeChannels* added to **OpenCVFilters** for interpolated size change of a data object
- timestamp added to tags of acquired data objects in **PGRFlyCapture**. Further fixes to achieve highest possible framerate. (don't use extended shutter for high frame rates and don't set the smallest possible frame time, since this leads to a really low frame rate, a frame time close to the minimum achieves high frame rates.)
- **Ximea**: bugfixes, added *gpo* and *gpi*, backward compatibility to older Ximea API...
- filter *cropBox* for **PclTools** inserted to crop points inside or outside of an arbitrary positioned and rotated box.
- **PclTools** adapted to PCL 1.8 and split into several files due to a big-object-compiler error.
- bugfixes in **glDisplay**, can now also display Rgba32 data objects. Additionally, existing textures can be changed with *editTextures*.
- many config dialogs and toolboxes inherit now the new abstract base classes.
- fixes in grabFramebuffer of **dispWindow** plugin
- fixes in cvUndistortPoints, cvInitUndistortRectifyMap and cvRemap (**OpenCVFilters**)
- fitPlane and getInterpolatedValues in **FittingFilters** can now also be executed using the "least median of squares"
- plugin **OpenCVFiltersNonFree** created. This contains non BSD licensed code from OpenCV and is not included in the itom setup per default.
- plugin **AVTVimba** created and released under LGPL (for cameras from Allied Vision)
- plugin **AndorSDK3** create and released under LGPL (for cameras from Andor, tested with Zyla 5 camera)
- plugin **NewportSMC100** added and released under LGPL to control actuators from Newport (SMC 100)
- plugin **libmodbus** added and released under LGPL. This supports the communication protocol *modbus* (based on libmodbus 3.1.2 from <https://github.com/stephane/libmodbus>)
- plugin **PI_GCS2** added and released under LGPL. This controls *Physik Instrumente* devices using the GCS2 command set (tested with E-753).
- plugin **demoAlgorithms** released under LGPL.
- plugin **SuperlumBS** added and released under LGPL (for Broadband swept light source).
- plugin **NI-DAQmx** for National Instruments DAQmx interface added and released under LGPL.

Version 1.3.0 (2014-10-07)

(more than 100 commits in plugins repository)

- plugin *Thorlabs CCS* for spectrometers from Thorlabs added (dataIO plugin). This plugin requires further drivers from the Thorlabs device.
- plugin *AerotechA3200* added to support the deprecated A3200 interface from company Aerotech.
- fixes in plugin *PIPiezoCtrl*: parameters *delayOffset*, *delayProp* and *async* are now really transmitted to the device (did nothing before)
- fixes in *PCOCamera* plugin with camera *PCO.1200s* that does not support the *setPoint* temperature.
- all plugins adapted for Qt4 and Qt5.
- plugin *dispWindow* adapted to OpenGL 3.1 and 4.0. Deprecated shader commands replaced. Parameters 'lut' and 'gamma' are now working and the gamma correction is enabled if parameter **gamma*=1*
- filter *cvUndistort* in *OpenCVFilters* can now handle every data type as input.
- fixes some bugs when importing csv files
- filter *cvFlipFilter* also supports multi plane flipping for 3D data objects.
- plugin *GLDisplay* added that allows displaying one or multiple arrays on the screen using OpenGL to provide a very fast flip between several images.
- many enhancements and improvements in plugin *pclTools* (mainly done by company twip optical solutions GmbH): filter for fitting spheres to point clouds added, filter to calculate distances to a given model added, filter to prepare a display of these distances added, methods partially OpenMP parallelized, filter for fitting cones to point clouds added, filter for projecting point clouds to models added.
- plugin *PGRFlyCapture* now runs under linux, general changes to support Grasshopper3 cameras (color supported as well).
- some fixes in plugin *cmu1394* and optional byte swapping for 16bit camera added
- improvements in camera plugin *Vistek*
- improved error handling when trying to load unsupported tiff formats
- filters *gaussianFilter* and *gaussianFilterEpsilon* added to plugin *BasicFilters*
- filters *cvRotate180*, *cvRotateM90* and *cvRotateP90* added to *OpenCVFilters*
- improvements and better synchronization to camera in plugin *Ximea*. Experimental shading correction added.
- bugfix when loading a x3p file -> yscale has not been loaded correctly
- camera plugin *IDS uEye* added to support cameras from company IDS Imaging (based on their driver 4.41)

Version 1.2.0 (2014-05-18)

(more than 200 commits in plugins repository)

- PCOCamera: improved plugin with renew config dialog and toolbox. Tested with PCO.1300 and PCO.2000.
- DataObjectIO: Added importfilter for ascii based images or point lists
- PCLTools released under LGPL
- PGRFlyCapture: extended shutter added for longer integration times, frame rate is not used if extended shutter is one since images are acquired as fast as possible.
- AerotechEnsemble: bugfixes: avoid timeout for long-time operations corrected status message when all axes are in-target position
- fixes for PCOPixelFly: some board IO errors are handled
- DispWindow adapted to OpenGL 3.x specification as well as Qt5
- Initial release of video 4 linux (V4L2)

- Improved SDF-Export function with invalid handling for MountainsMaps (plugin DataObjectIO)
- exec-function for ramp-trajectory added to GwInstek plugin (power supply controller with RS232 connection)
- Plugin FireGrabber supported under linux. FirePackage driver used under Windows, Fire4Linux under linux
- Plugin SerialIO under linux: both tty and usb ports supported
- Plugin x3pio released under LGPL (wrapper for data format x3p, see opengps.eu)
- Added clipping-Filters and history-filter to BasicFilters
- Many documentations for plugins added
- Plugin MSMediaFoundation released under LGPL (requires at least Windows Vista, successor of DirectShow for accessing basic cameras)
- Parameters sharpness and gamma added to Ximea plugin.
- Plugin LibUSB released.
- Fixed big-endian, little-endian bug in PointGrey plugin, parameters sharpness, gamma and auto_exposure added
- Plugin for PointGrey cameras released under LGPL.
- Plugin for XIMEA cameras released under LGPL.
- FFTW plugin added that is a wrapper to the FFTW library (GPL license!)
- overall modification of Vistek camera plugin: Toolbox, configuration dialog based on new base classes of itom SDK, better parameter handling, improved image grabbing...

Version 1.1.0 (2014-01-27)

there is no continuous changelog for these version

1.3 Designer Plugins

Version 2.0.0 (2015-07-20)

(more than 95 commits in designerPlugins repository)

- itom1dqwtplot, itom2dqwtplot: property unitLabelStyle added to chose how the labels print unit information
- line cut slices and z-stack slices can now also be displayed in a previously indicated line-plot widget (properties lineCutPlotItem and zSlicePlotItem)
- various styles possible for labels containing units (with respect to current standards)
- Qt4/Qt5 string encoding problems fixed (latin1 / utf8)
- dataObjectTable: more signals added (clicked, activated, pressed, doubleClicked, entered) to allow better access via python.
- slider2D: designer plugin added
- itom1dqwtplot: logarithmic and double-logarithmic scale added for x- and y-axis (base 2, 10 and 16)
- itom1dqwtplot and itom2dqwtplot: method *send current view to python workspace* added to put a shallow copy of the currently zoomed region of interest to the python workspace.
- itom1dqwtplot: histogram-like plots with the properties fill-color and fill-style added
- itom1dqwtplot: many more style properties for lines
- itom1dqwtplot and itom2dqwtplot: many fixes concerning resize, rescaling, representation in free and fixed-aspect-ratio mode.

- itom1dqwtplot: picker can now have labels. Many style properties set.
- itom2dqwtplot and itom1dqwtplot: Improvements to allow multi layer line plots
- itom1dqwtplot: color values of rgba-data objects will be displayed with three different colors
- pre-compiler symbols for windows, mac and linux unified
- support for Mac OS added (osx)
- itom2dqwtplot: z-slice can only be started at positions inside of the object
- vtk3dvisualizer: camera position and view can now be set, commands 'addText' and 'updateText' added, geometry items can now have properties like specular, specularPower, specularColor
- vtk3dvisualizer: support for Qt4/5 and VTK5/6
- itom2dqwtplot: non-finite values are displayed as transparent pixels

Version 1.4.0 (2015-02-17)

(more than 50 commits in designerPlugins repository)

- itom1dqwtplot: changed slots for setPicker / getPicker from ito::int32 / ito::float32 to int and float due to conversion / call problems.
- itom1dqwtplot: fixes some rescaling problems when switching the complex representation or the row/column representation.
- itom1dqwtplot, itom2dqwtplot: improvements in panner, magnifier, zoomer with and without fixed aspect ratio. Magnification is now possible using Ctrl + mouse wheel.
- itom1dqwtplot, itom2dqwtplot: geometric elements can now obtain labels (accessible via slots)
- Initial commit of vtk3dVisualizer to visualize pointclouds, polygon meshes, geometric elements. These elements are organized in a tree view and can be parametrized. The display is realized using Vtk and the PointCloudLibrary.
- Encoding fixes in itom1dqwtplot and itom2dqwtplot due to default encoding changes in Qt5 (switched from Latin1 to Utf8)
- itom2dqwtplot: Added property to change geometric element modification mode
- itom1dqwtplot: Improved linewidth for copy pasted export
- itom1dqwtplot, itom2dqwtplot: zoom stack of zoomer and magnifier tools is synchronized with panner such that changing the plane or complex representation does not change the zoomed rectangle after a panning event)
- itom1dqwtplot, itom2dqwtplot: some handling fixes in export properties of 1d and 2d qwt plot. The properties are now shown before the file-save-dialog in order to give the user an overview about the possibilities before he needs to indicate a filename.
- itom1dqwtplot, itom2dqwtplot: shortcuts added for actions 'save' and 'copyToClipboard'
- itom1dqwtplot: property lineStyle and lineWidth added
- itom1dqwtplot, itom2dqwtplot: copy-to-clipboard added to tools menu of 1d and 2d qwt plot. Improved keyPressEvent for both plots (playing with event->ignore() and event->accept())
- itom1dqwtplot: rounding fix in order to show the right data to given z-stack-cut coordinates.
- improvements in itom2dqwtplot: z-stack picker obtains the general color set including a semi-transparent background box; the z-plane can be selected via a property 'planeIndex'
- itom2dqwtplot: z-stack and linecut window has an appropriate window title
- itom1dqwtplot, itom2dqwtplot: Working on an improved geometric element handling (e.g. modes for move, modify points) Adapted type switches and comparisons to handle flagged geometric elements via type ito::PrimitiveContainer::tTypeMask
- itom1dqwtplot, itom2dqwtplot: Added new icons for geometric element modification.

- Added shift and alt modifier to `itom2dqwtplot` to move / rotate geometric lines with fixed length
- update to qwt 6.1.2 for compability with Qt 5.4
- Improving `EvaluateGeometricsFigure` to evaluate 3D-Data
- Improved functionality of `EvaluateGeometricsFigure` to calculate distances between ellipse centers
- fix in `itom1dqwtplot` and `itom2dqwtplot`: `dataObjects` were not updated if only their content, but not the size, type... changed
- changes for access of `plotItemChanged` via python
- Added `colorMap` to `overlayImage` for `Itom2dQwtPlot` via `overlayColorMap-Property`
- `itom1dqwtplot`: legend (optional) added to `itom1dqwtplot` (properties: `legendPosition` (Off, Left, Top, Right, Bottom) and `legendTitles` (StringList) added). Per default, the legend is not displayed, and if it is displayed, the default names are curve 0, curve 1, curve 2...
- `itom2dqwtplot` is principally able to display 1xN or Nx1 data objects (was blocked until now; but sometimes people want to do this)
- `itom1dqwtplot`, `itom2dqwtplot` adapted to `ito::AutoInterval` class (`xAxisInterval`, `yAxisInterval`, `zAxisInterval` are of this type now)
- `itom1dqwtplot`, `itom2dqwtplot`: Added background, axis and tick color

Version 1.3.0 (2014-10-07)

(more than 40 commits in `designerPlugins` repository)

- The properties `xAxisInterval` and `yAxisInterval` return the currently visible or set interval (bugfix)
- overlay image of `itom2dqwtplot` can now be read out by the property `overlayImage`
- Firsts steps for an auto-documentation of designer plugins
- Linecut of `itom2dqwtplot` can now also be set to the horizontal / vertical line containing the global minimum or maximum
- Some bugfixes concerning the display of `dataObjects` that are shallow copies from numpy arrays
- Fixes a bug that showed errors when a linecut and a z-stack-cut of `itom2dqwtplot` and a 3d data object is visible at the same time
- Mode for single and multi row or column display of `itom1dqwtplot` for 2d data objects as input
- Center marker in `itom2dqwtplot` can be adjusted in size and pen using the general style settings for designer plugins (`itom.ini` setting file only)
- improvements and rework of zoomer, panner and magnifier with or without fixed aspect ratio for `itom2dqwtplot` and `itom1dqwtplot`
- magnifier of `itom2dqwtplot` and `itom1dqwtplot` now also works with Ctrl+mousewheel
- `pickPoints` event now also works in `itom2dqwtplot` if the zoomed rectangle or the magnification is changed during interaction
- improved unit switches in GUI in `motorController`
- slots added for saving and rendering to pixmap for `itom2dqwtplot`
- property `grid` added to `itom1dqwtplot` to show/hide a grid
- some problems fixed with point selectors in `itom1dqwtplot`
- matplotlib is now a designer plugin based on `AbstractFigure` like other plots as well. It can then be docked into the main window.

Version 1.2.0 (2014-05-18)

(more than 80 commits in `designerPlugins` repository)

- `itom2DQwtPlot` is able to display color data objects and cameras.

- Press Ctrl+C to copy the currently displayed plot in *itom1DQwtPlot* and *itom2DQwtPlot* to clipboard. Also available via menu.
- display of pointClouds in *itomIsoGLFigurePlugin*
- fix in autoColor mode (*itom2DGraphicView*) with rgba32 data objects or cameras
- save dialog remembers last directory
- secondary dataObject can be plotted as semi-transparent overlay (alpha value adjustable) in *itom2DQwtPlot* (Python access via property *overlayImage*)
- many bugfixes
- multiline plotting in *itom1DQwtPlot* improved
- designer plugins are now ready for inclusion in GUIs of other plugins

Version 1.1.0 (2014-01-27)

there is no continuous changelog for these version

INTRODUCTION

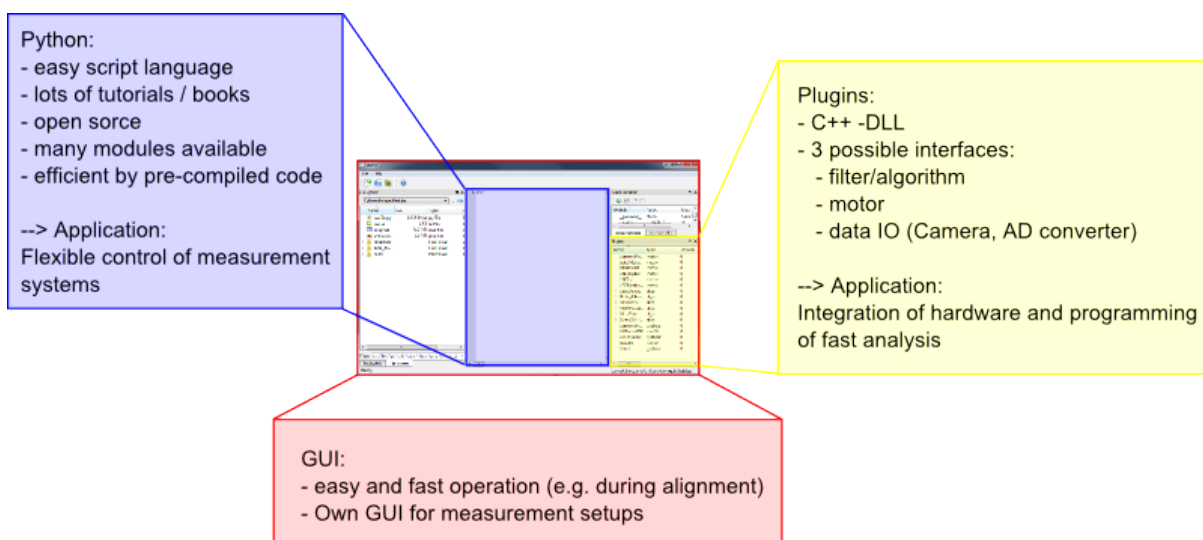
itom is a lab automation and measurement software developed, maintained and provided by the ITO (Institut for Technical Optics, University Stuttgart, Germany). While this software has been designed with an emphasis on developing and running optical systems, it is not limited to this field of application. **itom** represents a versatile tool for building the control software for any measurement setup aimed at a high degree of adaptability to different parameters and variable hardware. To address these requirements it was decided to design a powerful and fast core-program, integrating a scripting language in order to easily modify and transfer code blocks from one part of the program to another and to provide a simple and unitary interface for controlling external hardware components. Beyond the control of the physical setup, **itom** provides a wide variety of features for the analysis and processing of the acquired data. **itom** is best considered as an open source lab automation software whose functionalities lie somewhere between commercial software packages like Matlab or Labview.

The following table shows in which way base requirements to such a software system have influenced the choice of components during the designing process:

What we wanted	How we did it
fast software, development of fast algorithms	C++
modern multi-platform GUI	Qt-Framework
easy integration of different hardware (camera, actuator,...)	Plugin system
fast, robust and easy-to-learn scriptinglanguage	Python

In the figure below you can see the three columns on which **itom** is based:

1. Python
2. Plugins
3. GUI



Based on these three columns you can control measurement applications, basic setups or scripted image processing.

To learn more about how to control **itom** via script language or the GUI proceed to *Getting Started*.

TODO: Add the basic idea and structure of ITOM and explain it in more detail than this above

2.1 About us

Institut fuer Technische Optik
Universitaet Stuttgart
Pfaffenwaldring 9
70569 Stuttgart

Report issues at:

<https://bitbucket.org/itom/itom/issues>

This help has been built for itom-version 2.0.0 (SVN/Git-revision 8227a0e).

2.2 Licensing

2.2.1 itom Licensing

The core components and the main application of **itom** are covered by the **GNU Library General Public Licence** (GNU LGPL). All components belonging to the SDK of **itom** (e.g. *dataObject*, *pointCloud*, *addInInterface*,...) are additionally covered by an **itom exception**. The main idea of this exception is to allow other libraries (e.g. plugins) to include and link against components of **itom** SDK independent on the specific license model of the respective "other" library. All files belonging to the **itom** SDK are included in the folder **SDK** that is shipped with any setup or included in the build directory (when build from sources).

The full text license of **LGPL** and **itom exception** is also included as file *COPYING* in the source distributions and setups.

All plugins and designer-plugins that can be integrated into **itom** can have their own licensing. Therefore the user is referred to the specific licensing documents or statements of each external library (plugin).

2.2.2 itom Exception

All components (source files) which are part of **itom** SDK are not only covered by the LGPL license but also by the *itom Exception*, that provides further grants and rights for using these components:

```
ITO LGPL Exception version 1.0
-----

Additional rights granted beyond the LGPL (the "Exception").

As a special exception to the terms and conditions of version 2.0 of the LGPL,
ITO hereby grants you the rights described below, provided you agree to
the terms and conditions in this Exception, including its obligations and
restrictions on use.

Nothing in this Exception gives you or anyone else the right to change the
licensing terms of the itom Software.

Below, "Licensed Software" shall refer to the software licensed under the LGPL
```

and this exception.

- 1) The right to use Open Source Licenses not compatible with the GNU Library General Public License: Your software (hereafter referred to as "Your Software") may import the Licensed Software and/or distribute binaries of Your Software that imports the Licensed Software.
- 2) The right to link non-Open Source applications with pre-installed versions of the Licensed Software: You may link applications with binary pre-installed versions of the Licensed Software.
- 3) The right to subclass from classes of the licensed software: Classes that are subclassed from classes of the licensed software are not considered to be constituting derivative work and therefore the author of such classes does not need to provide source code for this classes.

2.2.3 Package Licences

A standard distribution of **itom** links to the following third-party library (shared linkage):

- [OpenCV](#) by Willow Garage under BSD-license
- [Point Cloud Library](#) by Willow Garage under BSD-license
- [QT-Framework](#) by Nokia under LGPL.
- [QScintilla](#) in by Riverbank Computed Limited under GPL with additional exceptions.
- QScintilla is a port to Qt of Neil Hodgson's [Scintilla](#) C++ editor control.
- [Python](#) by Python Software Foundation under Python-License (similar to BSD license).
- Python-package [NumPy](#) under BSD compatible license.
- Some classes concerning the interaction between **Qt** and **Python** have been inspired by [PythonQt](#) under LGPL license.

Additionally, the following third-party libraries are also used by common plugins or common configurations of **itom**:

- [SciPy](#) under BSD compatible license.
- [Matplotlib](#) under BSD compatible license.
- [QwtPlot](#) by Uwe Rathmann and Josef Wilgen under LGPL with additional [exceptions](#).
- [QPropertyEditor](#) under LGPL
- Google test framework by Google under New BSD-license

Other plugins may also contain further third party packages, e.g.:

- NVidia CUDA SDK by NVidia under NVidia CUDA SDK License
- [OpenMP](#)
- Hardware dependent third party drivers and SDKs with different license models or terms of conditions than the main itom-programm. So please check the specific plugin-license.

INSTALLATION

In this section we want to show you how to get and install **itom** on your computer.

3.1 Minimum system requirements

Before installation, please review the minimum system requirements.

- Operating System
 - Windows XP SP3 *or*
 - Windows Vista SP1 (SP2 and platform update recommended) *or*
 - Windows 7 *or*
 - Windows 8 (not tested yet) *or*
 - Linux based OS (tested with Debian and Ubuntu)
 - Mac OS X 10.9 (Mavericks) or later
- Processor architecture
 - Windows and Linux: 32 or 64bit processor architecture, SSE and SSE2 support
 - Max OS X: All Intel-based Macs with an Intel Core 2 or later
- 800MHz processor (dual-core recommended)
- 512MB of RAM
- 1024 x 768 screen resolution
- 200+ MB hard drive space

3.2 Installation from setup

Currently, there are both a *32bit* and a *64bit* setup version of **itom** available (Windows only). For linux users there is no pre-compiled package available. The use of the setup version is recommended for users that mainly want to work with **itom** without developing new algorithms or plugins. The setup can be downloaded from <https://bitbucket.org/itom/itom/downloads>. The setup installs the core application, a number of useful hardware plugins and the designer plugins which provide plotting functionalities to **itom**.

3.2.1 Windows Setup Installation

For your convenience you can download an installer package for Microsoft Windows from [<https://bitbucket.org/itom/itom/downloads>](https://bitbucket.org/itom/itom/downloads). There are several different types of setups available:

- Windows, 32bit, **itom** only

- Windows, 64bit, **itom** only
- Windows, 32bit, **itom**, Python + packages: numpy, scipy, matplotlib, PIL (optional)
- Windows, 64bit, **itom**, Python + packages: numpy, scipy, matplotlib, PIL (optional)

These setups do not contain any plugins or designer-plugins.

In the course of the installation, the following third-party components will be installed along with **itom**:

1. Microsoft Visual C++ 2010 Runtime Libraries (x86 or x64)
2. Python 3.2.3 (optional)
3. Python package *numpy* 1.6.2(optional)
4. Python package *scipy* 0.10.1 (optional)
5. Python package *matplotlib* 1.2.x (optional)
6. Python package *PIL* 1.1.7 (optional)

In the following we will guide you through the installation setup with a couple of screenshots:

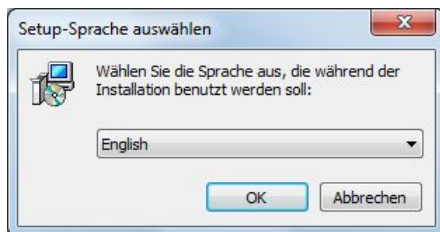


Fig. 3.1: Please select your desired language for the setup.

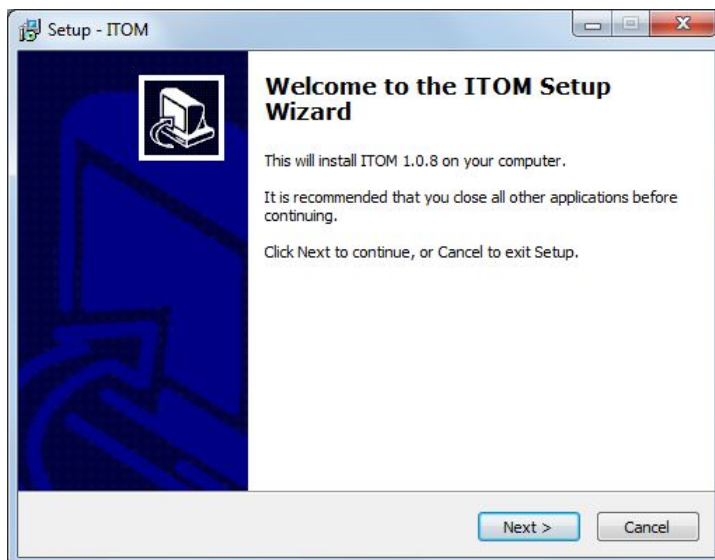


Fig. 3.2: The start screen of the setup will appear.

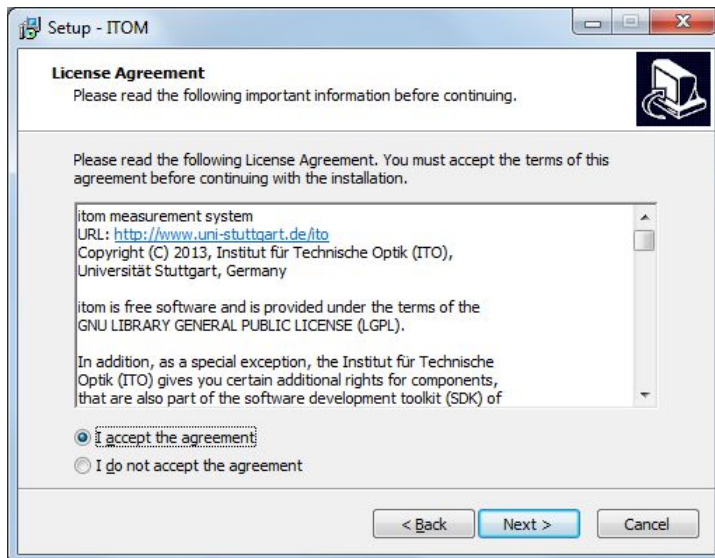
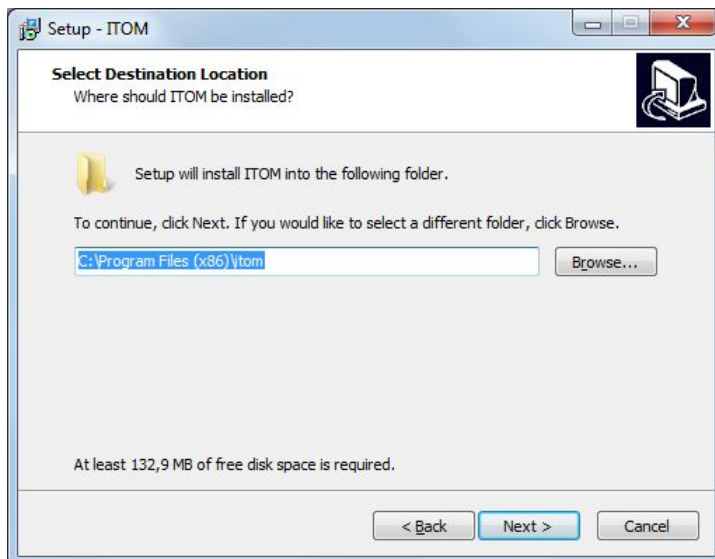


Fig. 3.3: Read the license text and agree to it.

Fig. 3.4: Choose where to install **itom** on your file system.

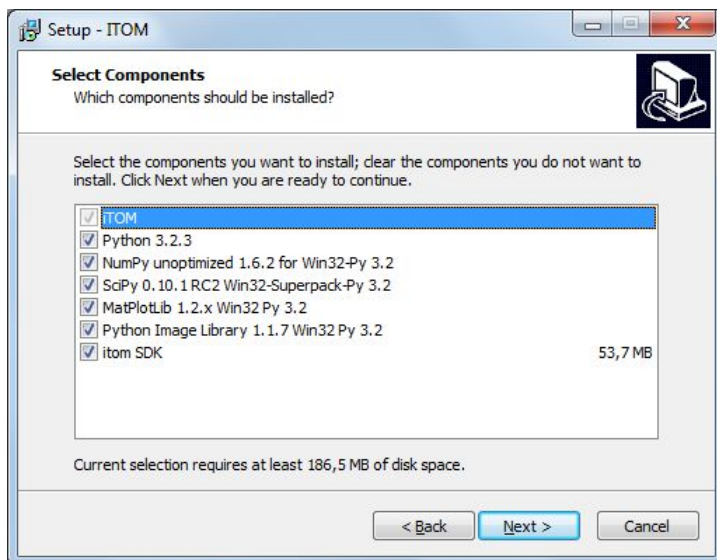


Fig. 3.5: Depending on your version of the setup, you now need to (de)select some optional components. The SDK is important if you want to develop your own plugins for **itom**. If you have the extended setup version, you can also select that python including some important packages is directly installed. This is only recommended, if you do not have python in a similar version already installed on your computer. You can also manually install and/or update python or its packages before or after this setup.

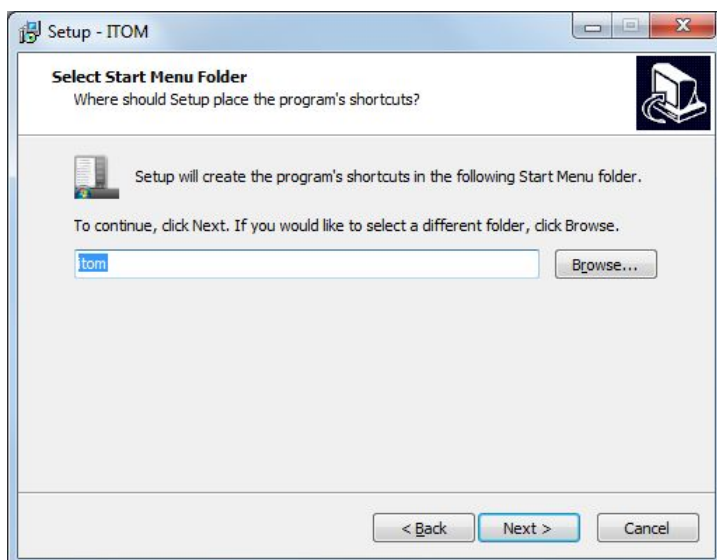


Fig. 3.6: Select the name of **itom** in your Windows start menu.

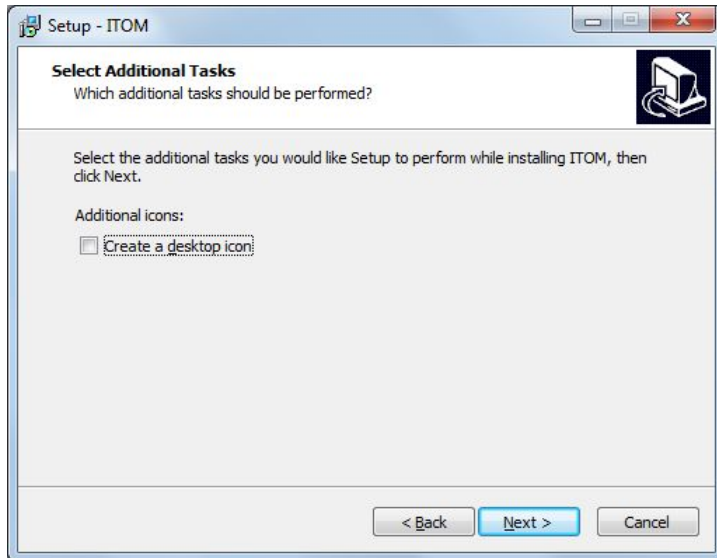


Fig. 3.7: Choose whether you want to have an **itom** shortcut on your desktop

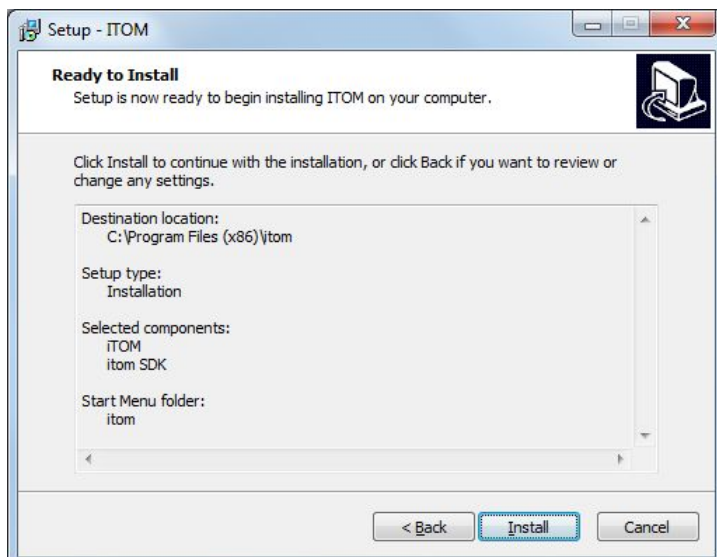


Fig. 3.8: Now, a summary of the installation steps is given. Press next if you want to start the installation...

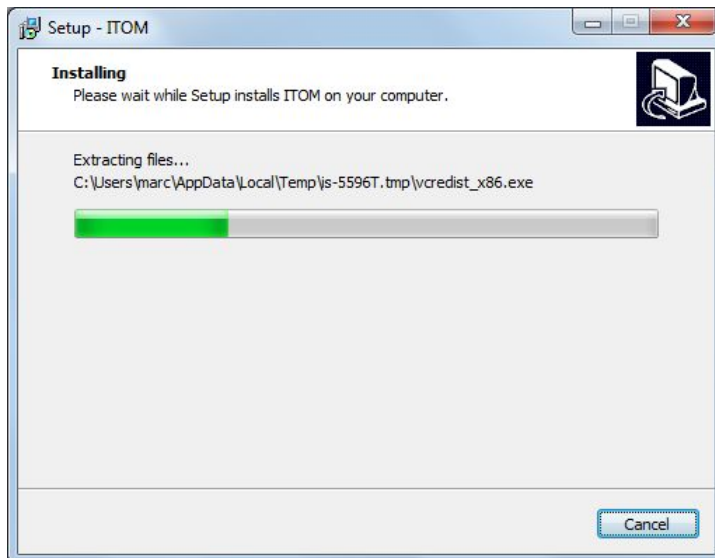


Fig. 3.9: The installation is executed now. **itom** is not copying any files in another folder than the indicated program folder (besides python or any python-packages). However this setup creates an application entry in the Windows registry in order to allow an uninstall by the default Windows control panel and to check if any version of **itom** already has been installed. When uninstalling **itom**, the registry entry is removed, too.

Depending on your selected components, python and/or any python packages are now installed:

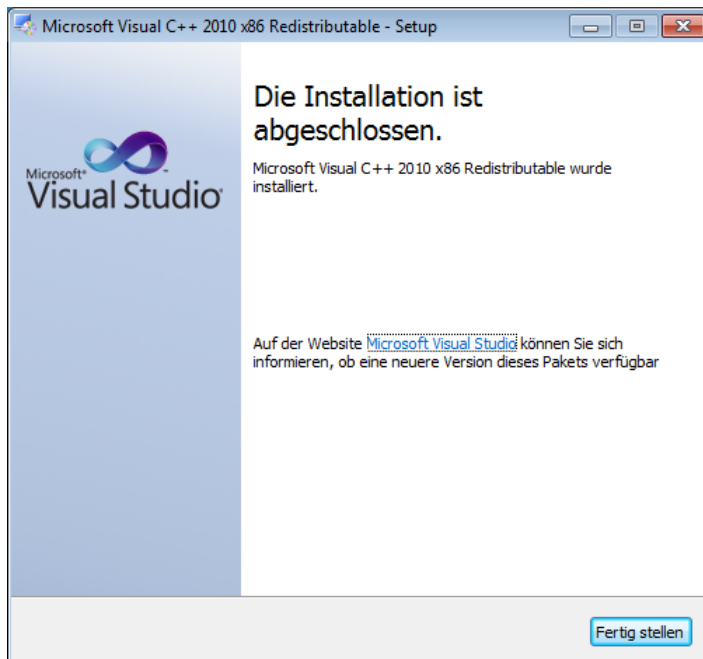


Fig. 3.10: If not already available, the Microsoft Visual C++ 2010 Runtime Libraries are installed now.

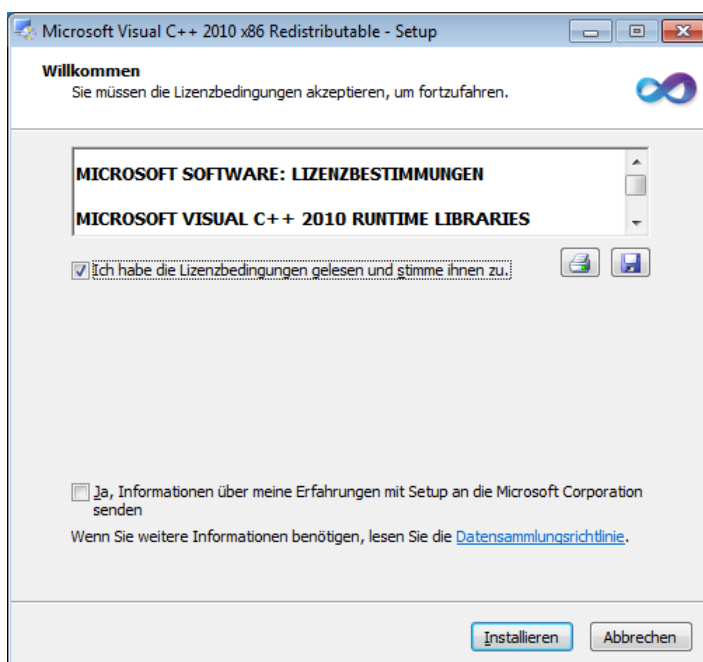




Fig. 3.11: Install Python 3.2.1 for the current user or for all users.



Fig. 3.12: Customize your Python installation. We recommend leaving everything as is.



Fig. 3.13: You've completed the Python installation as well. We're getting getting closer.

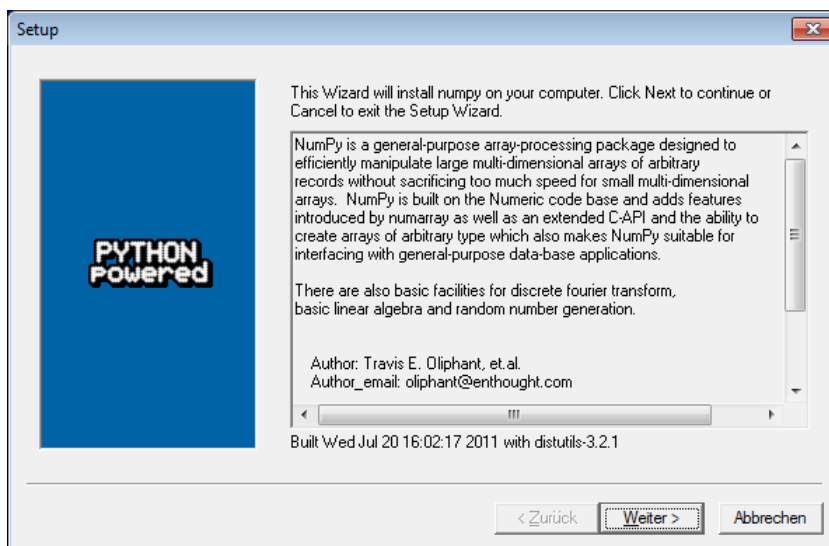


Fig. 3.14: Continue with the installation of NumPy.

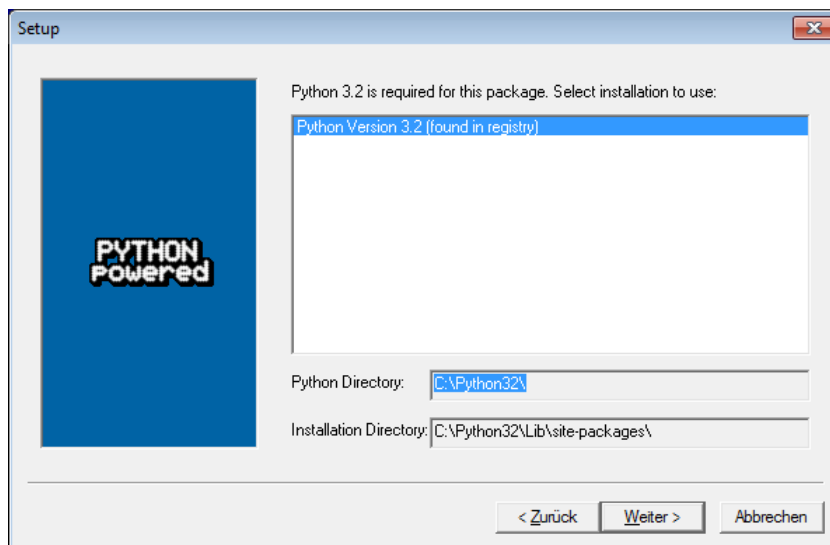


Fig. 3.15: You should be able to continue right away.

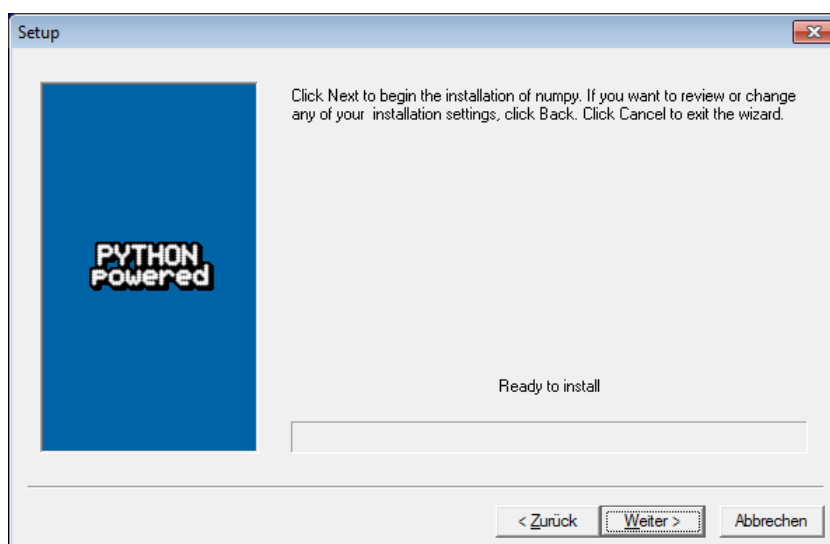


Fig. 3.16: Start the NumPy installation.

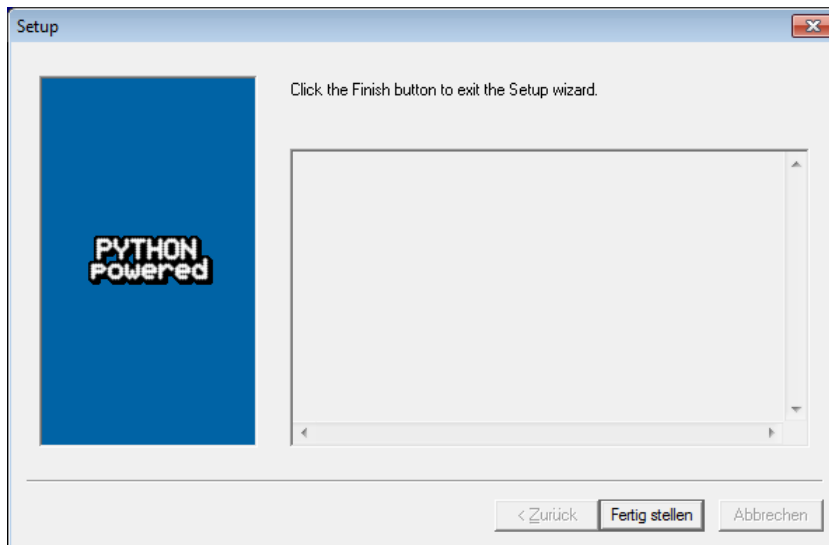


Fig. 3.17: Confirm that NumPy was successfully installed.



Fig. 3.18: Eventually, we have to install scipy, press “Next” to continue

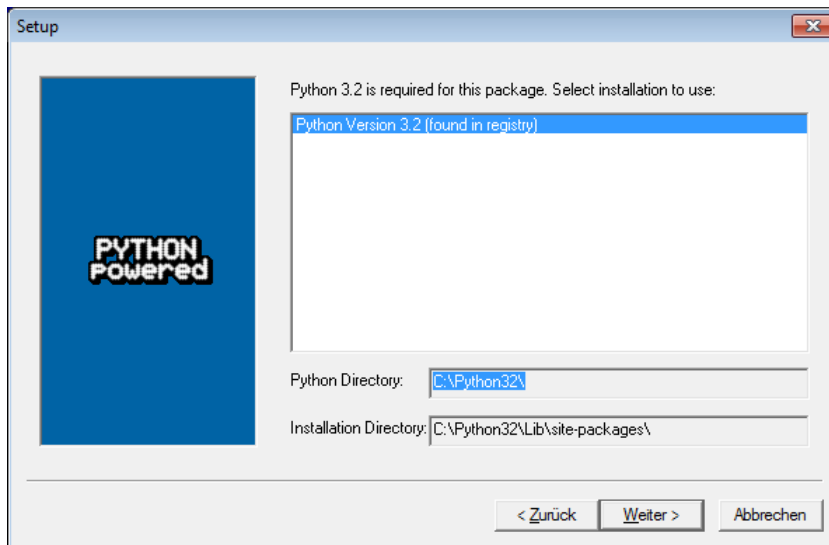


Fig. 3.19: Continue.

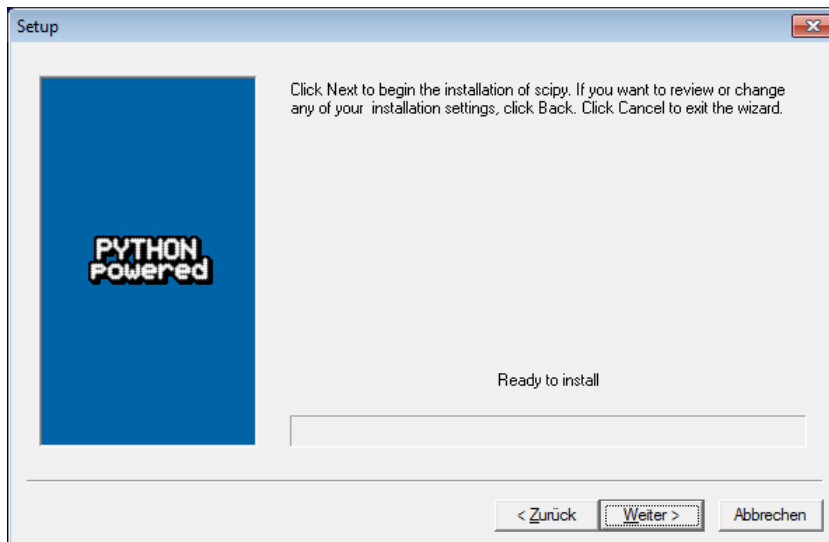


Fig. 3.20: Yes, we are ready to install. Proceed please.

Finally, the entire setup is finished:

That's it:

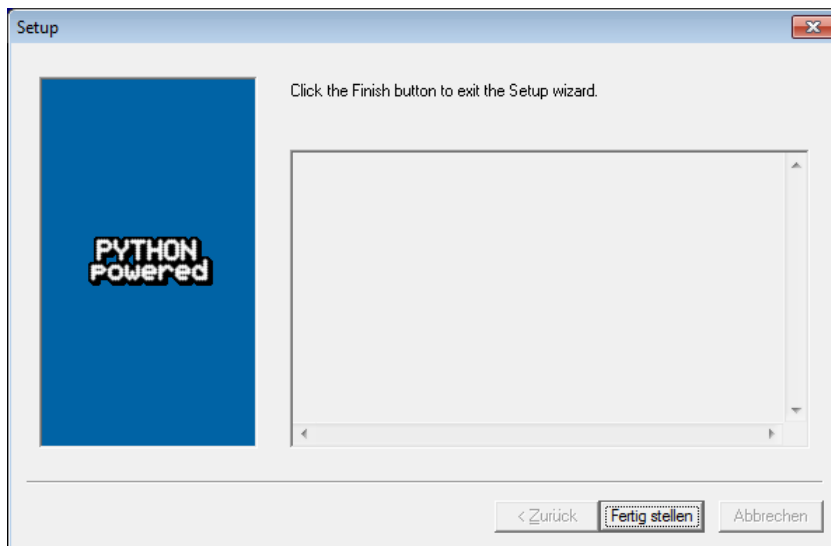
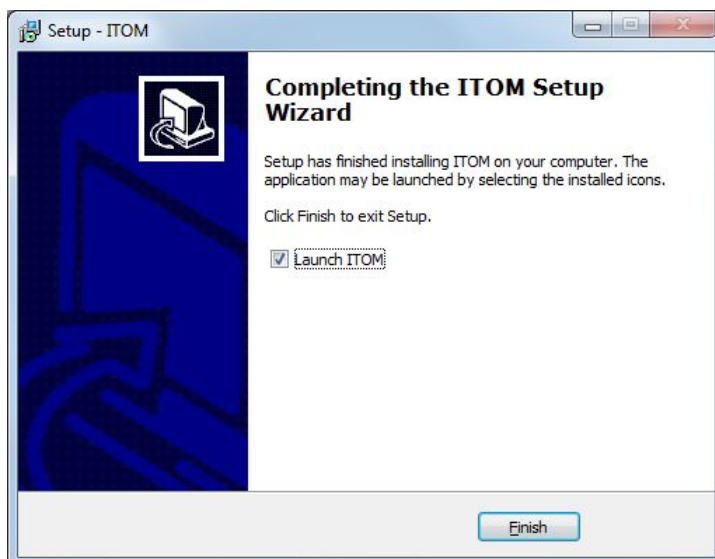
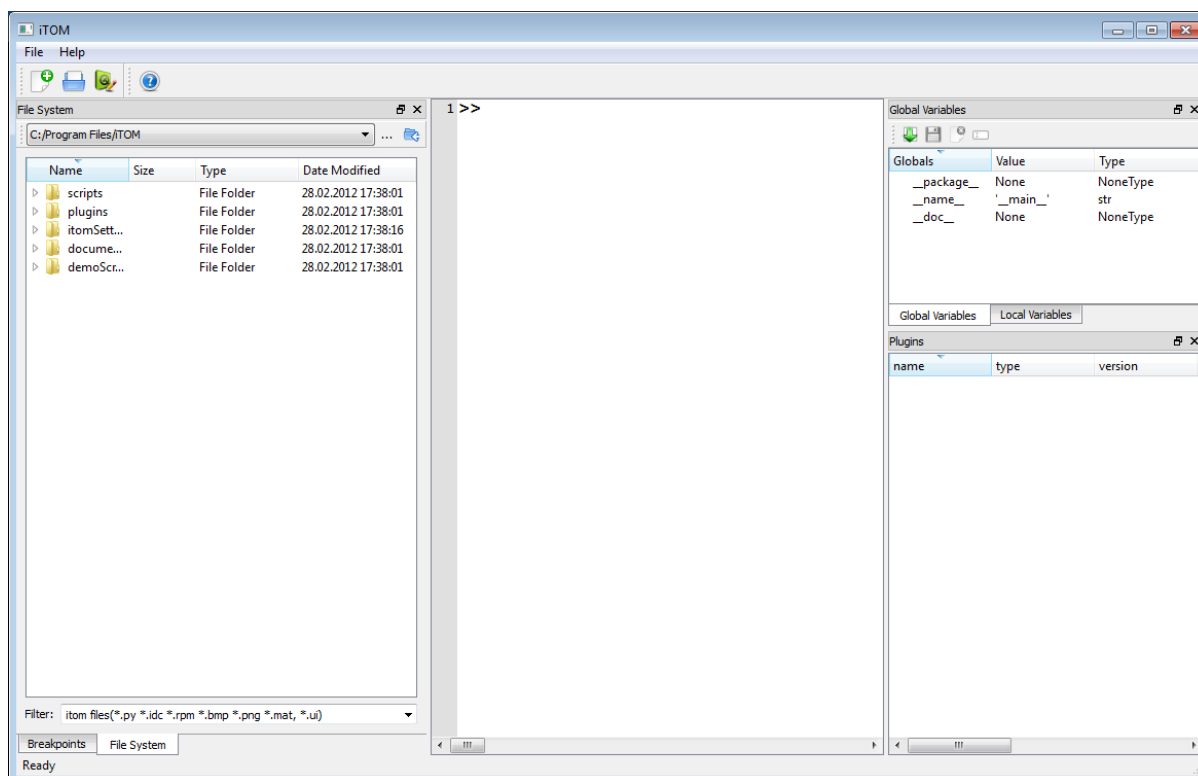


Fig. 3.21: Another confirmation, now scipy has been installed successfully.



Fig. 3.22: This is **itom**!

3.3 Build from Sources

Note: There is a all-in-one build development installation available (Windows, Visual Studio 2010 only). This contains all dependencies that are required to build itom and its main plugins from sources. See the section *All-In-One development setup* below for more information.

Alternatively, it is possible to get the sources of **itom** (e.g. clone the latest Git repository from <https://bitbucket.org/itom/itom.git>) and compile an up-to-date version of **itom**. This is recommended for developers (e.g. plugin developers) and required for linux users. Before getting the source files, check the build dependencies below which contain software packages and libraries necessary to build and **itom**.

3.3.1 Build dependencies

This setup lists third-party software packages and libraries that are required if you want to build **itom** from sources. If you run a setup-release of **itom** none of these dependencies (besides a python 3 installation) are required. Most of the following hints address the build on a Windows operating system. However the required packages are mainly the same for Linux and most components can directly be obtained by the specific package manager of your Linux distribution.

Software packages

Required Software-Packages

- IDE, Compiler (e.g. Visual Studio 2010 Professional, QtCreator...)
- CMake (recommended 2.8.9 or higher)

- Qt-framework (4.7 or higher, 4.8 or \geq 5.3 recommended)
- QScintilla2 (2.6 or higher)
- OpenCV 2.3 or higher (2.4 recommended)
- Python 3.2 or higher
- Git (git-scm.com) + GUI (e.g. TortoiseGit or GitExtensions) for accessing the remote repository
- Python-Package: NumPy

Optional Software-Packages

- PointCloudLibrary 1.6 or higher (\geq 1.7 recommended, optional)
- Qt-AddIn for Visual Studio (requires .NET 2.0 framework with SP 1.0)
- Doxygen (for creating the source code documentation)
- Python-Packages: SciPy, Distribute, Sphinx (user documentation generation), scikit-image, matplotlib...

Detailed information

Compiler, IDE (mandatory)

You can use any compiler and integrated development environment (IDE) which is supported by **CMake** (http://www.cmake.org/cmake/help/v2.8.10/cmake.html#section_Generators). On Windows systems, we develop with **Visual Studio 2010 Professional**, whereas we use **QtCreator** for the development under Linux. QtCreator is no specific CMake generator, however QtCreator directly supports CMakeLists.txt-files. It is also possible to use the free express edition of Visual Studio.

Note: Please consider that you need to install the Service Pack 1 for Visual Studio 2010 Professional when compiling a 64bit version of **itom**. It is even recommended to install the service pack for a 32bit compilation.

CMake (mandatory)

Download **CMake** from <http://www.cmake.org/cmake/resources/software.html> and install it. If possible use any version higher than 2.8.8. CMake reads the platform-independent project files of **itom** (CMakeList.txt) and generates the corresponding project files for your compiler, IDE and platform.

Qt-framework (mandatory)

Download the **Qt-framework** (version 4.7 or higher, 4.8.x or \geq 5.3 recommended) from <http://qt-project.org/downloads>. If you find a setup version for your IDE and compiler, you can directly install it. Otherwise, you need to configure and build Qt on your computer - see box below (e.g. Qt 4.8.x with Visual Studio 2010, 64bit needs to be compiled). For Qt5 either download the ready-to-use binaries from qt-project.org, compile it from sources and follow the instruction in the box below or consider to use the *all-in-one-development setup*. If you use the ready-to-use binaries, make sure to use a version with OpenGL.

Create the following environment variables (Windows only - you need to log-off from your computer in order to activate changes to environment variables):

- create an entry **QTDIR** and set it to the Qt-base directory (e.g. C:\Qt\4.8.0)
- for Qt4.x create an entry **QMAKESPEC** and set it to the string **win32-msvc2010** (even if you are compiling for 64bit) or similar (see <http://qt-project.org/doc/qt-4.8/qmake-environment-reference.html#qmakespec>)
- add the following text to the Path variable: **;%QTDIR%\bin** (please only **add** this string, **do not replace** the existing path-entry)

Note: Compiling Qt 4.7.x or 4.8.x (using the example 64bit, Visual Studio)

This side-note explains how to configure and build Qt for a 64bit build using Visual Studio 2010. The general approach for other configurations is similar.

- Delete files beginning with *sync* from the `%QTDIR%\bin` directory (in order to avoid the requirement of Perl during compilation, which is not necessary in our case).
- 64bit: Open **Visual Studio Commandline x64 Win64 (2010)** in your Start-Menu under *Microsoft Visual Studio >> Visual Studio Tools*.
- (for 32bit always use the **Visual Studio Commandline (2010)**)
- change to Qt-directory by typing:

```
cd %QTDIR%
```

into the command line.

- configure Qt-compilation by executing the command:

```
configure -platform win32-msvc2010 -debug-and-release -opensource -no-qt3support -qt-sql-odbc
```

- choose the option **open source version** and accept the license information during the configuration process. The configuration may take between 5 and 20 minutes.
- now start the time-intense compilation process (1 to 5 hours) by executing the command:

```
nmake
```

If you want to restart the entire compilation you need to completely remove any possible older configuration. Then open the appropriate Visual Studio command line and execute:

```
nmake distclean
```

Note: Compiling Qt 5.x (using the example 64bit, Visual Studio 2010)

This side-note explains how to configure and build Qt5 for a 64bit build using Visual Studio 2010. The general approach for other configurations is similar.

- Download the Qt5 sources a zip or tar.gz archive from qt-project.org and unpack them (e.g. to C:\Qt5.3.0)
- Make sure that Python 3.x is installed and verify that the path, containing the application **python.exe** is contained in the Windows environment variable.
- Open **Visual Studio Commandline x64 Win64 (2010)** (64bit) or **Visual Studio Commandline (2010)** (32bit) e.g. via *Windows >> Start >> Microsoft Visual Studio >> Visual Studio Tools*.
- Change to Qt-directory by typing:

```
cd %QTDIR%
```

- configure Qt by executing the command:

```
configure -platform win32-msvc2010 -debug-and-release -opensource -nomake tests -nomake examples
```

- choose the option **open source version** and accept the license information.
- now start the time-intense compilation process by executing:

```
nmake
```

- If ready type:

```
nmake install
```

- Finally, you can build the documentation (build into the qtbase/doc folder) by typing (see

```
nmake docs
```

If you want to restart the entire compilation you need to completely remove any possible older configuration. Then open the appropriate Visual Studio command line and execute:


```
nmake distclean
```

If Python could not be accessed, an error during the compilation may occur. Then make sure that Python is accessible via the Path environment variable and delete the possibly available file `C:/Qt/Qt5.3.0/qtdeclarative/src/qml/RegExpJitTables.h`.

Qt-Visual Studio-AddIn (optional, only for Visual Studio, not necessary for QtCreator)

If you want to have a better integration of **Qt** into **Visual Studio** (e.g. better debugging information for Qt-types like lists or vectors), you should download the **Qt-Visual Studio-AddIn** (1.2.x for Qt 5.x, 1.1.11 for Qt 4.8.x, 1.1.10 for Qt 4.7.x) from <http://qt-project.org/downloads#qt-other> and install it. Since we are using **CMake** it is not mandatory to use this **AddIn** like it is usually the case when developing any Qt-project with Visual Studio. Therefore it is also possible to use the Express edition of Visual Studio, where you cannot install this add-in. The **Qt Visual Studio AddIn** requires that you have the **.NET framework 2.0 SP 1** installed on your PC.

Note: Sometimes, there are problems when starting Visual Studio with an installed Qt-AddIn. In case that any component cannot be registered, as warned by a message-box when starting Visual Studio, you should check the bug and its fix described at <https://bugreports.qt-project.org/browse/QTVSADDINBUG-77>. In most cases it was sufficient to register the library **stdole.dll** using the tool **gacutil.exe** from the **Microsoft SDKs/Windows/v7.0A/bin** subfolder of your standard program folder. Start a windows commandline and move to the directory on your computer where the executable program **gacutil.exe** is located, then type:

```
gacutil.exe -i "C:\Program Files (x86)\Common Files\microsoft shared\MSEnv\PublicAssemblies\stdole.dll"
```

QScintilla2 (mandatory)

Download **QScintilla** (2.6 or higher) from <http://www.riverbankcomputing.com/software/qscintilla/download> and build the debug and release version. The original QtCreator project file of QScintilla finally copies the entire output to the **bin** directory of **Qt** so that no other settings need to be adapted. However, the original project settings are not ready for a multi-configuration build in **Visual Studio**. As a result, you need to adapt the Qt-project file. To do this follow these steps:

- Copy the downloaded files to a directory of your choice (preferably **NOT** the windows program directory, we are assuming in the following that you placed them in **C:\QScintilla2**)
- Open the Visual-Studio 2010 32-bit commandline (or 64-bit if installed)
- Open the file **C:\QScintilla2\Qt4\QScintilla.pro** or **C:\QScintilla2\Qt4Qt5\QScintilla.pro** (version 2.8 or higher) in a text editor and replace the line **CONFIG** with:

```
CONFIG += qt warn_off debug_and_release build_all dll thread
```

and add the lines:

```
CONFIG(debug, debug|release){ TARGET = $$join(TARGET,,d) }
```

(see also: http://www.mantidproject.org/Debugging_MantidPlot)

- If there is an error saying that the environment variable **QTDIR** is missing, type:

```
SET QTDIR=path-to-the-bin-directory-of-qt (do not add bin to the path itself)
```

- If you had a previous installation of QScintilla, delete the directory **%QTDIR%\include\Qsci** as well as the files called **qscintilla2.dll** and **qscintilla2d.dll** in the directory **%QTDIR%\bin**
- Execute the following commands from the command-line:

```
- cd C:\\QScintilla2\\Qt4
- nmake distclean
- %QTDIR%\\bin\\qmake qscintilla.pro spec=win32-msvc2010
- nmake
- nmake install
```

- copy the library files **qscintilla2.dll** and **qscintilla2d.dll** from **%QTDIR%\\lib** to **%QTDIR%\\bin**

Note: This is for ITO only (internal use): An easier approach is to get the sources from `\Obelix\software\m\ITOM\Installationen\4. QScintilla2` and copy the folder **QScintilla2.8** to a directory on your hard drive (e.g. `C:\QScintilla2.8`, avoid Windows program directory due to restrictions in write access). Open your Visual Studio Command Line and change to the directory of **QScintilla** on your hard drive. Just execute the batch file `qscintilla_install.bat` and answer the given questions.

Note: If the batch file breaks with some strange error messages, please make sure, that the location where **QScintilla** is installed is writable by the batch file (e.g. the Windows program directory under Windows Vista or higher). Therefore it is recommended to locate **QScintilla** in another directory than the program-directory.

OpenCV (mandatory, 2.3 or higher, 2.4.x recommended)

You have different possibilities in order to get the binaries from OpenCV:

1. Download the OpenCV-Superpack (version 2.3) from <http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.3/>. This superpack is a self-extracting archive. Unpack it. The superpack contains pre-compiled binaries for VS2008, VS2010, MinGW in 32bit and 64bit. (Later map the CMake variable **OpenCV_DIR** to the **build** subdirectory of the extracted archive).
2. Download the current setup (version 2.4 or higher, recommended) from <http://opencv.org/> and install it. This installation also contains pre-compiled binaries for VS2008, VS2010 and MinGW. In this case map **OpenCV_DIR** to the **opencv/build** subdirectory.
3. Get the sources from OpenCV and use CMake to generate project files and build the binaries by yourself. Then map **OpenCV_DIR** to the build-directory, indicated in CMake.

Finally, add the appropriate bin-folder of OpenCV to the windows environment variable: - VS2010, 32bit: Add to the path-variable: `;%C:\OpenCV2.3\build\x86\vc10\bin` (or similar) - VS2010, 64bit: Add to the path-variable: `;%C:\OpenCV2.3\build\x64\vc10\bin` (or similar)

Changes to the environment variable only become active after a re-login to windows.

Note: There is a known linker problem with OpenCV 2.4.7 (only this version). Please avoid to use this special version.

PointCloudLibrary (optional, 1.6 or higher)

The PointCloud-Library is a sister-project of OpenCV and is able to work with large point clouds. You can compile **itom** with support for the point cloud library. Then the python classes **itom.pointCloud**, **itom.point** and **itom.polygonMesh** are available and algorithm plugins can use point cloud functionalities. If you don't need anything like this, don't install the point cloud library and uncheck the option **BUILD_WITH_PCL** in the CMake configurations of **itom**.

The binaries can be loaded from the website <http://www.pointclouds.org/downloads/windows.html>. Depending on 32bit or 64bit execute the **AllInOne-Installer for Visual Studio 2010**. The installation directory may for example be `C:\PCL1.6.0`. Information: Please install the PCL base software including all 3rd-party packages, besides OpenNI. You don't have to install OpenNI, since this is only the binaries for the communication with commercial range sensors, like Kinect.

If you want to debug the point cloud library (not necessary, optional) unpack the appropriate zip-archive with the pdb-files into the bin-folder of the point cloud library. This is the folder where the dll's are located as well.

Add the path to the bin-folder of PointCloud-library to the windows environment variable:

- Add to the path-variable: `;%C:\PCL1.6.0\bin` (or similar)

Python (mandatory, 3.2 or higher)

Download the installer from <http://www.python.org/download/> and install python in version 3.2 or higher. You can simultaneously run different versions of python.

NumPy (mandatory)

Get a version of NumPy that fits to your python version and install it. On Windows, binaries for many python packages can be found under <http://www.lfd.uci.edu/~gohlke/pythonlibs/>.

pip (optional)

Pip is the new package installation tool for **Python** packages. If you don't have **pip** already installed (already included in Python >= 3.4) use the following hints to get **pip**. Download the file from <https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py> and save it to any temporary directory. Then open the file **get-pip.py** with the python version used for compiling **itom** (e.g. python32.exe). As an alternative, open a command line and switch to the directory where you save the file **get-pip.py**.

Assuming that Python is located under **C:\Python32**, execute the following command:

```
C:\python32\python.exe get-pip.py
```

pip is installed and you can use the **pip** tool (see **Sphinx** installation above).

Sphinx (optional)

The Python package **Sphinx** is used for generating the user documentation of **itom**. You can also download sphinx from <http://www.lfd.uci.edu/~gohlke/pythonlibs/>. However, sphinx is dependent on other packages, such that it is worth to install Sphinx using the **Python** tool **pip** (If you don't have **pip** see the next section). Then open a command-line (cmd.exe) and switch to the directory **[YourPythonPath]/Scripts**. Type the following command in order to download **sphinx** including dependencies from the internet and install it:

```
pip install sphinx
```

For upgrading **sphinx**, type:

```
pip install sphinx --upgrade
```

frosted (optional)

The Python package **frosted** can be installed in order to enable a code syntax checker in **itom**. If installed, your scripts are automatically checked for syntax errors that are marked as bug symbols in each line. The detailed messages are displayed as tool tip texts of the bug symbol. Use pip to install this package:

```
pip install frosted
```

Other python packages (optional)

You can always check the website <http://www.lfd.uci.edu/~gohlke/pythonlibs/> for appropriate binaries of your desired python package.

Note: If you use any python packages depending on NumPy (e.g. SciPy, scikit-image...) try to have corresponding versions. If your SciPy installation is younger than NumPy, some methods can not be executed and a python error message is raised, saying that you should update your NumPy installation.

3.3.2 Get sources from Git

The core of **itom**, some plugins as well as designer-plugins are hosted in different Git-repositories on *bitbucket.org*.

GIT is a distributed version and source code management system, that allows several developers to work on the same source code simultaneously. **itom** is hosted under GIT at bitbucket.org.

The different parts can be found at the following resources:

Project	Website	Git-Repository (https)	Git-Repository (ssh)
itom (core)	https://bitbucket.org/itom/itom	https://bitbucket.org/itom/itom.git	git@bitbucket.org:itom/itom.git
plugins	https://bitbucket.org/itom/plugins	https://bitbucket.org/itom/plugins.git	git@bitbucket.org:itom/plugins.git
designer- Plugins	https://bitbucket.org/itom/designerPlugins	https://bitbucket.org/itom/designerPlugins.git	git@bitbucket.org:itom/designerPlugins.git

In order to get the sources you want, clone the specific repository to a source-folder on your computer. If you want to receive data by *ssh*, you need your own account on *bitbucket.org*, where you public *ssh* has been set under your personal settings.

3.3.3 Build with CMake

In this chapter, you will learn how to create the project files from the sources, contained in the specific Git-repositories (*Get sources from Git*), using the open source generator **CMake** (<http://www.cmake.org>). The main concept of CMake will be introduced in the following section, followed by a short explanation of process of building the core of **itom**.

Concept of CMake

CMake is a platform independent make system for C or C++ based software projects. Therefore the sources only contain the code base and some simple textfiles (e.g. CMakeLists.txt) that describe how a project must be configured such that the code base can correctly be compiled.

When calling the CMake GUI, the configuration files (CMakeLists.txt) will be analyzed. After indicating your desired compiler or generator of your preferred development system (see http://cmake.org/cmake/help/v2.8.8/cmake.html#section_Generators), CMake will generate the specific makefiles or project files for your generator. Then you can call your development environment or compiler like Visual Studio in order to open the recently created project files and finally build your application.

Experienced Visual Studio user are used to locating the source files (.h, .cpp...), the visual studio project files, the intermediate object files and the final compiled application in the same folder structure. This is different for projects that are initiated using CMake, since this process is closer to the base philosophy of Linux.

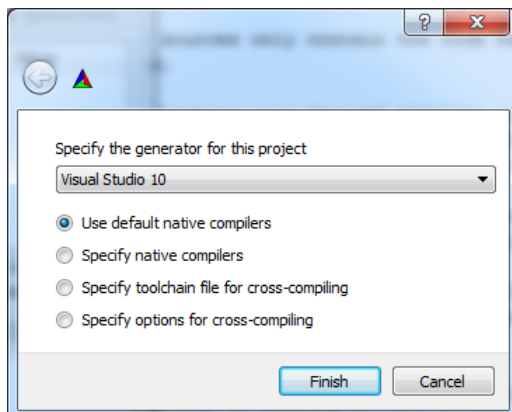
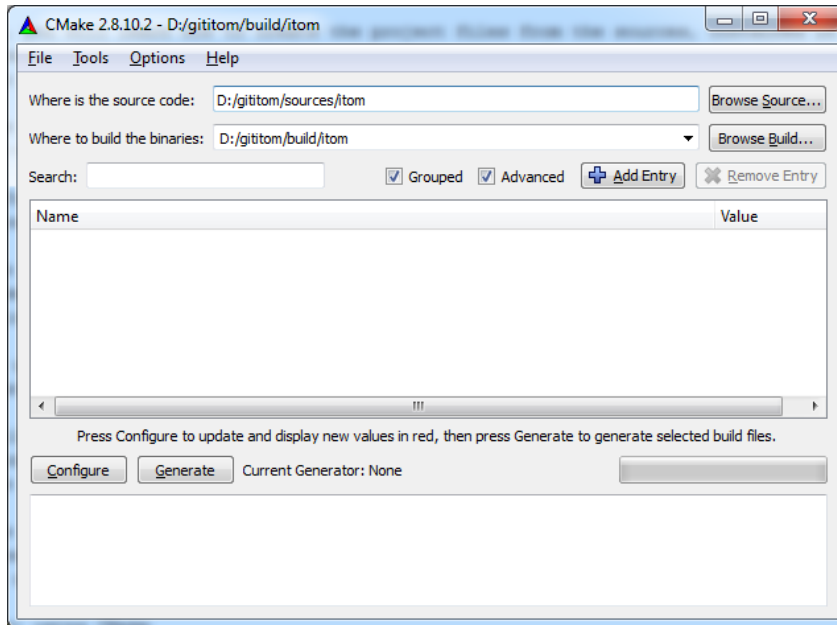
In CMake you will have one folder that only contains the code base, some further resource files and the general CMakeLists.txt-files. CMake is called to analyze this source-folder and to generate the project files in another, independent folder, that is denoted build-directory in the following text. You will only work in this build-directory, that is also the directory, where the intermediate files as well as the compiled executable are located.

All itom plugins can have one or more own build-directories, such that the code base for some or all plugins can also be totally delocated from itom itself. When generating the project files for any plugin using CMake as well, you are asked where the build-directory of itom is located on your harddrive. This is finally used in order to copy the relevant output of every plugin to this itom application directory in a post-build step of every plugin or designer-plugin.

Generating itom using CMake

These are the steps you need to execute if you want to generate and build **itom** from sources using CMake:

1. Get the sources of **itom** core from the Git repository (see *Get sources from Git*).
2. Start **CMake GUI**.
Use the **browse source...** button in order to select the location of the itom source code, e.g. *D:/gititom/sources/itom*. Use **browse build...** to indicate the build directory for itom, e.g. *D:/gititom/build/itom*. If this directory does not exist, it will be created during the configuration.
3. Now click the **Configure** button in CMake. Then, the file *CMakeLists.txt* in your source folder is analyzed. If you create this project for the first time, you are now asked for your desired generator.
If you compile using Visual Studio, choose **Visual Studio 10** or **Visual Studio 10 Win64**. For the **QtCreator** choose **MinGW Makefiles** on Windows or **Unix Makefiles** on Linux machines. Accept your choice by clicking **Finish**.
4. Since **itom** is dependent on other libraries (like OpenCV), CMake is able to automatically guess the location for many common libraries. The resulting variables including many other settings for configuring the compiler will then be shown in the huge listbox in CMake. If CMake added a variable for the first time, this entry is displayed using a red background. Clicking **Configure** again, the red background becomes white.



- Usually, the configuration process will fail when clicking the button for the first time, since you need to help CMake with some manual changes. The detailed problem is always written in the log-text at the bottom part of the CMake GUI. Help CMake with some further hints (see below) and press **Configure** again. The current configuration is finally cached in a specific cache-file, that also is located in the build directory such that any reconfiguration will firstly read the content of this cache file.
- Click the **Grouped** checkbox, in order to get a better overview of the current configurations. Further variables are shown when clicking the **Advanced** checkbox. Usually you need to manually set all or some of the following variables (in this order):

Variable	Description
OpenCV_DIR	Set this variable to the directory of OpenCV. This is the build directory contained in the OpenCV base directory, if you used the precompiled binaries or the directory where you built the OpenCVbinaries using CMake itself. Reclick Configure after having set OpenCV_DIR to the correct value. If OpenCV could be located, more variables starting with OpenCV will appear in the CMake GUI.
PCL_DIR	Only set this variable if its value is different than PCL_DIR_NOTFOUND . This variable must point to a directory similar to <i>C:/PCL/PCL1.6.0/cmake</i> .
BUILD_SHARED	click this checkbox (default: clicked) in order to use the shared libraries of Qt which is needed if the LGPL-version of Qt is used.
BUILD_TARGET64	click this checkbox if you want to build itom for 64bit. In the future, this should be automatically detected.
BUILD_UNICODE	click this checkbox (default: clicked) if you want to compile itom with unicode strings.
CMAKE_BUILD_TYPE	is to one of the strings given in CMAKE_CONFIGURATION_TYPES . For Visual Studio, this is unimportant, since Visual Studio configures your project for all possible configurations. With QtCreator see http://lists.qt.nokia.com/public/qt-creator/2009-November/005011.html .
VISUAL-LEAKDETECTOR_DIR	you can only set this value if compiling itom with Visual Studio in Debug mode. Point it to the directory similar to <i>D:\itomtrunk\Visual Leak Detector</i> . This folder must contain subfolders named bin, include and lib. Click VISUALLEAKDETECTOR_ENABLED in order to enable the memory leak detector in Visual Studio. Please make sure, that you add the correct subfolder of its bin directory to the windows environment variables or copy the content to the executable directory of itom (where qitom.exe is finally located).

- relick **Configure** until no red-backgrounded values are visible and no errors appear. Then click **Generate** to create the project files.
- If you are using Visual Studio, open it now and load the solution **itom.sln**, contained in the build directory. This solution now contains several sub-projects. All projects besides **ALL_BUILD** and **ZERO_CHECK** are itom specific sub-projects. The easiest way to compile itom is to mark qitom as start-project (right click on qitom and select as start project) and build **qitom**. The dependent projects **dataobject**, **pointcloud** and **qpropertyeditor** are automatically build before building qitom. The project **ALL_BUILD** will firstly start **ZERO_CHECK** and then build all projects contained in the solution in the dependent-optimal order. For more information about **ZERO_CHECK** see the Update project settings section below.
- If you are using **QtCreator**, open it and call the file *CMakeLists.txt* from the itom source folder. Additionally chose the same output folder defined in the CMake GUI. QtCreator can now read the cached configurations from CMake. The **QtCreator** will also run CMake, a process that should be safe and should not fail, as everything should have been properly configured in the previous steps using CMake GUI. Usually you don't need the CMake GUI with **QtCreator**, however you can not easily change any settings directly in **QtCreator**.

Generating any plugin or designer-plugin using CMake

In principle, the build of plugins or designer-plugins follows the same basic steps than the build of **itom** itself. However, both plugins and designer-plugins need some files and libraries, that are contained in the **SDK** of **itom**. If you installed **itom** from a setup, the **SDK** is already available, if you built **itom** from sources, you first need

to compile **itom** (at least the libraries *dataObject*, *pointCloud* and *qpropertyeditor*) in **Debug and Release** before continuing to configure and to generate any plugin project.

In some cases, multiple plugins or designer-plugins are bundled together in one main directory. You have the choice to either build one project with multiple sub-projects (which is a convenient way) or to build one project for each plugin. If you want to generate all plugins, directly use the main directory as source directory in CMake. If you only want to generate single plugins, directly use the corresponding subfolder as source directory. Any source directory must always contain a file *CMakeLists.txt*.

The most convenient way is to use the overall *CMakeLists.txt* file and choose single plugins by clicking the checkboxes of variables starting with *PLUGIN_xxx*.

When configuring any plugin or designer-plugin for the first time, you will probably see that either the variable **ITOM_SDK_DIR** or **OPENCV_DIR** is missing (or both), meaning that their values are set to **ITOM_SDK_DIR-NOTFOUND** or **OPENCV_DIR-NOTFOUND**. First set the variable **ITOM_SDK_DIR** to the **SDK**-directory of **itom**. This is either contained in the installation path of **itom** created by the setup or in the build-directory of **itom** if built from sources. Afterwards set **OPENCV_DIR** to the build-directory of **OpenCV** (see [this document](#) for details where the build-directory of **OpenCV** is). If one of these variables is not correctly set, other depending errors may be displayed in **CMAKE**, like:

```
CMake Error at AMMS/CMakeLists.txt:25 (find_package):
By not providing "FindOpenCV.cmake" in CMAKE_MODULE_PATH this project has
asked CMake to find a package configuration file provided by "OpenCV", but
CMake did not find one.
```

If the path to **ITOM_SDK_DIR** is set correctly and you pressed **Configure** again, the variable **ITOM_DIR** should also contain a valid folder (it should map to the build directory of **itom**, containing folders *plugins* and *designer*).

Update CMake

If you want to change any project settings or want to add or delete some source files from any project, then change the corresponding *CMakeLists.txt* files. If you need to change further settings, you should open the CMake GUI and re-configure your project. When done, press the **Generate** button. If Visual Studio is currently opened with the same project, the change in the project files is automatically recognized and you will be asked if you want to reload the project. Accept this. If you only changed some *CMakeLists.txt* files, you can also run the project **ZERO_CHECK** in Visual Studio. Visual Studio will call CMake in the command line in order to check for changes. The rest is the same as calling the CMake GUI. **QtCreator** is also able to directly run CMake in order to check the project for changes.

Additional third-party libraries

Some plugins are dependent on additional third-party libraries. The **itom** SDK provides some find-methods for several commonly used libraries. You will be informed by CMake if any library could not be found by this automatic find-method, if a variable of the following style and content is available:

```
LIBRARYNAME_DIR = LIBRARYNAME_DIR_NOTFOUND
```

LIBRARYNAME is then replaced by the real name of any missing library. See the following table for some common names **LIBRARYNAME**:

Library-name	What to do?
FFTW	Set FFTW_DIR to the directory of your fftw installation. This directory must contain some library files (e.g. libfftw3-3, libfftw3f-3...)
GLEW	Set GLEW_DIR to the directory where the binaries of glew are located on your computer.

FFTW

Some plugins of **itom** are using the library **fftw** in version 3. This library is always used as shared library. Do the following steps in order to get **FFTW 3** (on Windows):

1. Download the windows binaries (32bit or 64bit, depending on itom) from <http://www.fftw.org/install/windows.html>.
2. Unpack the binaries (e.g. at **C:fftw3.3**)
3. Now open the Visual Studio Command Line:
 - If you are compiling with 32bit, go to **Microsoft Visual Studio 2010 >> Visual Studio Tools >> Visual Studio-Command Line (2010)** in the Windows start-menu and open it.
 - If you are compiling with 64bit, go to **Microsoft Visual Studio 2010 >> Visual Studio Tools >> Eingabeaufforderung von Visual Studio x64 Win64 (2010)** in the Windows start-menu and open it.
 - In the opened command line, change to the FFTW directory, e.g. by typing **cd C:fftw3.3** and execute the following commands:

```
lib /def:libfftw3-3.def
lib /def:libfftw3f-3.def
lib /def:libfftw3l-3.def
```

4. Close the command line

If you are generating project files for itom plugins or designer plugins, you probably will sometimes get a FFTW-group. Set `FFTW_DIR` to **C:fftw3.3** and press configure again. All dependent files should be found.

GLEW

For Windows users:

Download the glew binaries from <http://glew.sourceforge.net/index.html> and unzip them to any folder. The CMake variable `GLEW_DIR` should then point to that folder.

Known problems during CMake configuration

- **Qt5 could not be detected:** Sometimes, Qt5 cannot be automatically detected. Then try to pass a directory similar to **G:\Qt\qtbase\lib\cmake\Qt5** to `Qt5_DIR`.
- **Linker error: Multiply defined symbols in msvcrt and similar libraries:** Make sure that OpenCV is not statically linked against itom. Make sure that `BUILD_OPENCV_SHARED` is True.

For linux as well as Mac OS X, a short description of the installation that contains more specific information than the sections above, is available here:

3.3.4 Build on linux

This section describes how **itom** and its plugins are built on a linux system (tested on Ubuntu 12.04 (32bit), Kubuntu (Debian, KDE, 64bit) and Lubuntu). The general approach is similar to the other documentation in the install chapter that are mainly focussed on Windows.

Necessary packages

Most necessary packages can be obtained by the package manager of your solution (Synaptic Package Manager, command *sudo apt-get...*). The following list describe packages that are required or recommended for building **itom**:

Required:

- **Qt4 or Qt5** (libqtcore4, libqt4-dev, libqt4-...)
- Editor **QScintilla** (libqscintilla2-8 or similar, libqscintilla2-dev)

- **OpenCV** (libopencv-core2.3 or libopencv-core2.4, libopencv-core-dev, libopencv-imgproc, libopencv-highgui...)
- **Python3** (python3, python3-dev, python3-dbg)
- **Numpy** (python3-numpy, python3-numpy-dbg)
- **git** (git)
- **Cmake** (cmake, cmake-gui)

Recommended (optional):

- The IDE **QtCreator** (qtcreator)
- **PointCloudLibrary** (if exists version 1.6 or better 1.7, else see <http://pointclouds.org/downloads/linux.html> or build it on your own, the point cloud library is optional!)
- **Scipy** (python3-scipy, python3-scipy-dbg), **Sphinx** (python3-sphinx), **Matplotlib**
- **Doxygen** (doxygen, doxygen-gui)
- Any git client (e.g. SmartGit (requires java) or git-cola)
- glew (libglew1.6-dev or something similar, required by some plugins)
- fftw (libfftw3-dev or something similar, required by some plugins)

for Ubuntu 12.04 based distributions the following command should install all necessary packages and its dependencies:

```
sudo apt-get install git libqt4-dev libopencv-dev libopencv-highgui-dev python3-dev python3-dbg qt4-dev-tools
```

Recommended folder structure

Similar to Windows, the following folder structure is recommended:

```
./sources
  ./itom      # cloned repository of core
  ./plugins   # cloned sources of plugins
  ./designerPlugins # cloned sources of designerPlugins
  ...
./build_debug # build folder for debug makefiles of...
  ./itom      # ...core
  ./plugins   # ...plugins
  ...
./build_release # build folder for release makefiles of...
  ./itom      # ...core
  ...
```

Under linux, the debug and release versions are separated in two different build folders. If you are using QtCreator as IDE you can however create two different configurations, one mapping to the debug build folder, the other mapping to the release build folder. Both use the same source folder.

Obtain the sources

Clone at least the core repository of **itom** (bitbucket.org/itom/itom) as well as the open source plugin and designerPlugin repository into the corresponding subfolders of the **sources** folder. You can do this by using any git client or the command **git clone https://bitbucket.org/itom/itom**.

Configuration process

Use **CMake** to create the necessary makefiles for debug and/or release:

1. Indicate the folder **sources/itom** as source folder

2. Indicate either the folder **build_debug/itom** or **build_release/itom** as build folder. If the build folder already contains configured makefiles, the last configuration will automatically loaded into the CMake gui.
3. Set the following variables:
 - **CMAKE_BUILD_TYPE** to either **debug** or **release**
 - **BUILD_TARGET64** to ON if you want to build a 64bit version.
 - **BUILD_UNICODE** to ON if you want to build with unicode support (recommended)
 - **BUILD_WITH_PCL** to ON if you have the point cloud library available on your computer and want to compile **itom** with support for point clouds and polygon meshes.
4. Push the configure button
5. Usually, CMake should find most of the necessary third-party libraries, however you should check the following things:
 - OpenCV: OpenCV is located by the file **OpenCVConfig.cmake** in the directory **OpenCV_DIR**. Usually this is automatically detected in **usr/share/OpenCV**. If this is not the case, set **OpenCV_DIR** to the correct directory and press configure.
 - Python3: On some linux distributions, CMake always finds the Python version 2 as default version. This is wrong. Therefore set the following variables to the right pathes: **PYTHON_EXECUTABLE** to **/usr/bin/python3.2**, **PYTHON_INCLUDE_DIR** to **/usr/include/python3.2**, **PYTHON_LIBRARY** to **/usr/lib/libpython3.2mu.so.1.0**. The suffix 1.0 might also be different. It is also supported to use any other version of Python 3.
6. Push the configure button again and then generate.
7. Now you can build **itom** by the **make** command or using **QtDesigner**:
 - **make**: Open a command line and switch to the **build_debug/itom** directory. Simply call **make** such that the file **qitom** or **qitomd** (debug) is built. Start this application in order to run **itom**.
 - **QtCreator**: Open QtCreator and open a new project. Indicate the file **CMakeList.txt** of **sources/itom** as project file. Now QtCreator asks where to build the binaries. Indicate the existing and pre-configured directory **build_debug/itom** or **build_release/itom**. The existing CMake cache files will be read and you can simply run CMake from QtCreator that should not fail. If so, re-open CMake and fix it. In the project settings of QtCreator you can finally clone the current configuration and indicate the second build-folder for the release version.

Build plugins

Build the plugins, **designerPlugins...** in the same way than **itom** but make sure that you compiled **itom** at least once before you start configuring and compiling any plugin. In CMake, you need to indicate the same variables than above, but you also need to set the variable **ITOM_DIR_SDK** to the **sdk** folder in **build_debug/itom/sdk** or **build_release/itom/sdk** depending whether you want to compile a debug or release version (please don't forget to set **CMAKE_BUILD_TYPE**).

If you don't want to have some of the plugins, simply uncheck them in CMake under the group **Plugin**.

The plugins and **designerPlugins** will finally be compiled and then copy their resulting library files into the **designer** and **plugins** subfolder of **itom**. Restart **itom** and you the plugin will be loaded.

3.3.5 Build on Mac OS X

This section describes how **itom** and its plugins are built on a Apple Mac systems running OS X 10.7 or later (tested on OS X 10.9 and 10.10). The general approach is similar to the other documentation in the install chapter that are mainly focussed on Windows.

Xcode

Xcode is the default IDE on OS X systems. It is also necessary to install Xcode to obtain the command line tools (including the compiler clang and the linker ld).

Since OSX Lion, the Xcode installation doesn't by default include the command line tools. Part of which a script called `easy_install` we will need to install some packages.

Once Xcode has been installed, run Xcode and then open the preferences. Select the Download section and the Components tab and install Command Line Tools from there.

CMake

Go to <http://www.cmake.org> to download and install the current stable release of CMake. It is recommended to download a release that looks like **cmake-x.y.z-Darwin-x86_64.dmg**

Homebrew

Most necessary packages can be obtained by the package manager of your solution. We will use Homebrew.

Open a terminal window and copy and paste the following line into a terminal

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Run the following command once before installing anything

```
brew doctor
```

Homebrew can also be installed as described at <http://brew.sh>.

Dependencies

The following list describe packages that are required or recommended for building **itom** and how to install them using brew.

Required packages:

- **Qt**
- **QScintilla2**
- **git**
- **Cmake**

Recommended packages:

- **PointCloudLibrary**
- **Doxygen**
- **glew**
- **fftw**

Required Python packages:

- **NumPy**

Recommended Python packages:

- **SciPy**
- **Pillow**
- **Matplotlib**

- sphinx
- frosted

Note: You will find the script `osx_install_dependencies.sh` in the source directory. This script allows to install all dependencies in one rush. To use it run the command `sh osx_install_dependencies.sh` in a Terminal.

To install all in one rush run the following list of commands. This might take a whole lot of time (we might talk about hours).

```
brew update
brew tap homebrew/science
brew tap homebrew/python
brew install git gcc python3 pkg-config
brew install qt --with-developer --with-docs
brew install pyqt --with-python3
brew install doxygen --with-doxywizard --with-graphviz
brew install qscintilla2 --with-python3
brew install ffmpeg glew fftw
brew install numpy --with-python3
brew link numpy --overwrite
brew install pillow --with-python3
brew link pillow --overwrite
brew install matplotlib --with-python3 --with-pyqt
brew link matplotlib --overwrite
brew install matplotlib-basemap --with-python3
brew link matplotlib-basemap --overwrite
brew install scipy --with-python3
brew link scipy --overwrite
brew install opencv pcl caskroom/cask/brew-cask
brew cask install qt-creator
brew linkapps
```

The above commands in one line:

```
brew update; brew tap homebrew/science; brew tap homebrew/python; brew install git gcc python3 pkg-config
```

If you would like to compile **itom** with Qt 5 replace `brew install qt --with-developer --with-docs` with `brew install qt5 --with-developer --with-docs` and `brew install pyqt --with-python3` with `brew install pyqt5 --with-python`.

Python 3

The default Python version on OS X is 2.x. Since **itom** is using Python 3.x you installed in the previous step but it is'nt recommended to replace version 2.x with 3.x. We will set an alias for python3, so when entered python in a terminal session, python3 will be called.

To edit you aliases execute the following command.

```
printf "alias python='python3'\n" >> ~/.bash_profile
```

The same thing must be done for *pip* and *easy_install*. Be advised to check the installed version number of python and change it when necessary. The command `python --version` will give you the installed version number.

```
printf "alias easy_install='/usr/local/Cellar/python3/3.4.3/bin/easy_install-3.4'\n" >> ~/.bash_profile
printf "alias pip='/usr/local/Cellar/python3/3.4.3/bin/pip3.4'\n" >> ~/.bash_profile
. ~/.bash_profile
```

Now we need to install two packages using *easy_install* (again remeber to check your version number!):

```
sudo easy_install virtualenv pyparsing frosted
```

We need some more python packages. Just run the following command:

```
pip install ipython mpmath sphinx
```

Recommended folder structure

Similar to Windows, the following folder structure is recommended:

```
./sources
  ./itom      # cloned repository of core
  ./plugins   # cloned sources of plugins
  ./designerPlugins # cloned sources of designerPlugins
  ...
./build_debug # build folder for debug makefiles of...
  ./itom      # ...core
  ./plugins   # ...plugins
  ...
./build_release # build folder for release makefiles of...
  ./itom      # ...core
  ...
```

To create all folders in your user directory in one step, call the following bash commands:

Obtain the sources

Clone at least the core repository of **itom** (bitbucket.org/itom/itom) as well as the open source plugin and designerPlugin repository into the corresponding subfolders of the **sources** folder. You can do this by using any git client or the command

Configuration process

Use **CMake** to create the necessary makefiles for debug and/or release:

1. Indicate the folder **sources/itom** as source folder
2. Indicate either the folder **build_debug/itom** or **build_release/itom** as build folder. If the build folder already contains configured makefiles, the last configuration will automatically loaded into the CMake gui. 2b. Check Advanced to *see* all available options. 3. Set the following variables:

- **Itom Parameter:**

- **BUILD_TARGET64** to ON.
- **BUILD_UNICODE** to ON if you want to build with unicode support (recommended)
- **BUILD_WITH_PCL** to ON if you have the point cloud library available on your computer and want to compile **itom** with support for point clouds and polygon meshes.

- **System Parameter:**

- **CMAKE_BUILD_TYPE** to either **debug** or **release**
- **CMAKE_OSX_ARCHITECTURES**: *x86_64*
- **CMAKE_OSX_SYSROOT**: */Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer*

The suffix SDK version might also be different as well as the path.

4. Push the configure button
5. Usually, CMake should find most of the necessary third-party libraries, however you should check the following things:
 - **OpenMP**: OpenMP is not available when using the default compiler clang. If your setup includes OpenMP set *-fopenmp* to *OpenMP_CXX_FLAGS* and *OpenMP_C_FLAGS*.

- **Python3:** On OS X CMake always finds the Python version 2 as default version. This is wrong. Therefore set the following variables to the right paths:
 - **PYTHON_EXECUTABLE:** /usr/local/bin/python3.4
 - **PYTHON_INCLUDE_DIR:** /usr/local/Cellar/python3/3.4.3/Frameworks/Python.framework/Versions/3.4/include
 - **PYTHON_LIBRARY:** /usr/local/Cellar/python3/3.4.3/Frameworks/Python.framework/Versions/3.4/lib/libpython3.4.dylib
 - **PYTHON_LIBRARY_DEBUG:** /usr/local/Cellar/python3/3.4.3/Frameworks/Python.PYTHON_LIBRARY.framework/Versions/3.4/lib/libpython3.4.dylib

The suffix 3.4 might also be different. It is also supported to use any other version of Python 3.

- **OpenCV:**
 - **OpenCV_BIN_DIR:** ../bin
 - **OpenCV_CONFIG_PATH:** /usr/local/share/OpenCV
 - **OpenCV_DIR:** /usr/local/share/OpenCV
- **QScintilla2:**
 - **QSCINTILLA_INCLUDE_DIR:** /usr/local/Cellar/qscintilla2/2.8.3/include
 - **QSCINTILLA_LIBRARY:** debug;/usr/local/Cellar/qscintilla2/2.8.3/lib/libqscintilla2.dylib;optimized;/usr/local/Cellar/qscintilla2/2.8.3/lib/libqscintilla2.dylib
 - **QSCINTILLA_LIBRARY_DEBUG:** /usr/local/Cellar/qscintilla2/2.8.3/lib/libqscintilla2.dylib
 - **QSCINTILLA_LIBRARY_RELEASE:** /usr/local/Cellar/qscintilla2/2.8.3/lib/libqscintilla2.dylib

The version 2.8.3 might also be different. It is also supported to use any other version of QScintilla2.

- **PointCloudLibrary (optional):**
 - **PCL_COMMON_INCLUDE_DIR:** /usr/local/include/pcl-1.7
 - **PCL_COMMON_LIBRARY:** /usr/local/lib/libpcl_common.dylib
 - **PCL_COMMON_LIBRARY_DEBUG:** /usr/local/lib/libpcl_common.dylib
 - **PCL_COMMON_DIR:** /usr/local/share/pcl-1.7

The version 1.7 might also be different. It is also supported to use any other version of the PointCloudLibrary.

6. Push the generate button.

7. Now you can build **itom** by the **make** or using **Xcode** command:

- **make:** Open a command line and switch to the **build_debug/itom** or **build_release/itom** directory. Simply call **make** such that the file **qitom** or **qitomd** (debug) is built. Start this application by calling **./qitom** or **./qitomd** in order to run **itom**.
- **Xcode:** If you plan to change something in the source code it is recommended to use Xcode. Just open the **itom.xcodeproj** with Xcode and compile the project.

Build plugins

Build the plugins, designerPlugins... in the same way than **itom** but make sure that you compiled **itom** at least once before you start configuring and compiling any plugin. In CMake, you need to indicate the same variables than above, but you also need to set the variable **ITOM_DIR_SDK** to the **sdk** folder in **build_debug/itom/sdk** or **build_release/itom/sdk** depending whether you want to compile a debug or release version (please don't forget to set **CMAKE_BUILD_TYPE**).

If you don't want to have some of the plugins, simply uncheck them in CMake under the group **Plugin**.

The plugins and designerPlugins will finally be compiled and then copy their resulting library files into the **designer** and **plugins** subfolder of **itom**. Restart **itom** and you the plugin will be loaded.

Known Problems

PyPort bug

If you get build errors that trace back to an error like

You are using a deprecated version of PyPort.

Locate **pyport.h**, it might be located in **/Library/Frameworks/Python.framework/Versions/3.4/include/python3.4m/pyport.h**. Open it and replace

```
#ifndef _PY_PORT_CTYPE_UTF8_ISSUE
#include <ctype.h>
#include <wctype.h>
#undef isalnum
#define isalnum(c) iswalnum(btowc(c))
#undef isalpha
#define isalpha(c) iswalalpha(btowc(c))
#undef islower
#define islower(c) iswlower(btowc(c))
#undef isspace
#define isspace(c) iswspace(btowc(c))
#undef isupper
#define isupper(c) iswupper(btowc(c))
#undef tolower
#define tolower(c) towlower(btowc(c))
#undef toupper
#define toupper(c) towupper(btowc(c))
#endif
```

with

```
#ifndef _PY_PORT_CTYPE_UTF8_ISSUE
#ifdef __cplusplus
/* The workaround below is unsafe in C++ because
 * the <locale> defines these symbols as real functions,
 * with a slightly different signature.
 * See python issue #10910
 */
#include <ctype.h>
#include <wctype.h>
#undef isalnum
#define isalnum(c) iswalnum(btowc(c))
#undef isalpha
#define isalpha(c) iswalalpha(btowc(c))
#undef islower
#define islower(c) iswlower(btowc(c))
#undef isspace
#define isspace(c) iswspace(btowc(c))
#undef isupper
#define isupper(c) iswupper(btowc(c))
#undef tolower
#define tolower(c) towlower(btowc(c))
#undef toupper
#define toupper(c) towupper(btowc(c))
#endif
#endif
```

See also <http://bugs.python.org/review/10910/diff/8559/Include/pyport.h>

python3 not available in Terminal

To make Python 3.x and its tools available to the command line add it to the PATH with a command like

```
sudo export PATH=/Library/Frameworks/Python.framework/Versions/3.4/bin:$PATH
```

Be sure to adapt the path as necessary. Especially be use to change the version number to the actually installed one.

pip install fails with `TypeError: unordered types: str() < NoneType()`

Open `/usr/local/Cellar/python3/3.4.3/libexec/setuptools/__init__.py` and change

```
self.py_version,  
self.platform,
```

for

```
self.py_version or '',  
self.platform or '',
```

Do the same for `/usr/local/lib/python3.4/site-packages/setuptools-12.2-py3.4.egg/pkg_resources/__init__.py`

Source: <https://bitbucket.org/minrk/setuptools/commits/e7d8f68ea7c603638562cf8278daa5d16e699f4e>

3.4 Plugins, Designer-Plugins

Each plugin or designer plugin enhances the core-functionality of **itom** and is compiled in its own project. Therefore the installer or sources of **itom** do not contain any plugin. Every plugin is distributed as a library file (*dll*, *so*,...) and - if necessary - other files.

3.5 All-In-One development setup

For users who want to get a development environment for itom, the main plugins and designer plugins there is an all-in-one development setup available.

Using this setup, you only need to unzip one or two archives to your harddrive, install **git** and **Python** that are included in this archive and execute a setup script, written in Python. This script automatically downloads the current sources of **itom** and its plugins from the internet, configures the 3rd party dependencies (also provided in this package) and automatically configures and generates CMake for the single repositories. Using this setup tool, you can start developing **itom** or plugins within a short time.

For more information see:

3.5.1 All-In-One development setup

For users who want to get a development environment for itom, the main plugins and designer plugins there is an all-in-one development setup available.

Using this setup, you only need to unzip one or two archives to your harddrive, install **git** and **Python** that are included in this archive and execute a setup script, written in Python. This script automatically downloads the current sources of **itom** and its plugins from the internet, configures the 3rd party dependencies (also provided in this package) and automatically configures and generates CMake for the single repositories. Using this setup tool, you can start developing **itom** or plugins within a short time.

The all-in-one development setup comes with the following features and 3rd party packages:

- Available for Visual Studio 2010 32bit and 64bit (Visual Studio 2010 and Service Pack 1 required)
- Git 1.9.4 (setup in package, needs to be installed)
- Python 3.4.2 (setup in package, needs to be installed)

- Numpy MKL 1.8.2 (setup in package, needs to be installed)
- Qt 5.3.2 (prebuild)
- OpenCV 2.4.10 (prebuild for VS2010)
- CMake 3.0.2 (prebuild)
- QScintilla 2.8 (prebuild inside of Qt 5.3.2)
- Doxygen

Optionally there is a 3rd party package that brings support for the PointCloudLibrary for itom. This secondary archive contains the following features:

- Boost 1.57.0 (prebuild)
- Eigen 3.0.5 (prebuild)
- Flann 1.7.1 (prebuild)
- QHull 2011.1 (prebuild)
- VTK 6.1.0 (prebuild with Qt 5.3.2 support)
- PCL 1.8.0 (prebuild with support of all libraries above)

Prerequisites for the development setup

If you want to use the development setup the following prerequisites must be fulfilled:

- Windows 7 or higher (XP untested)
- Visual Studio 2010 + Service Pack 1 must be installed on the computer

Get and install the setup

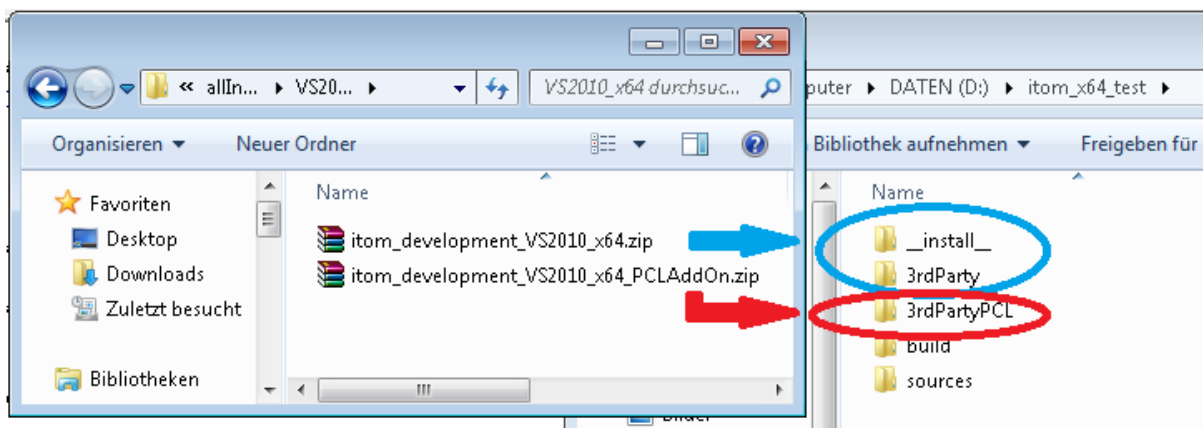
The setup comes with one or two zip-archive files:

- **itom_development_VS2010_x86.zip** or **itom_development_VS2010_x64.zip** (required)
- **itom_development_VS2010_x86_PCLAddOn.zip** or **itom_development_VS2010_x64_PCLAddOn.zip** (optional, for compilation itom with PointCloudLibrary support, 3D visualization...)

Download the 32bit or 64bit version depending on your needs from <https://sourceforge.net/projects/itom/files/all-in-one-build-setup/>.

Then execute the following steps:

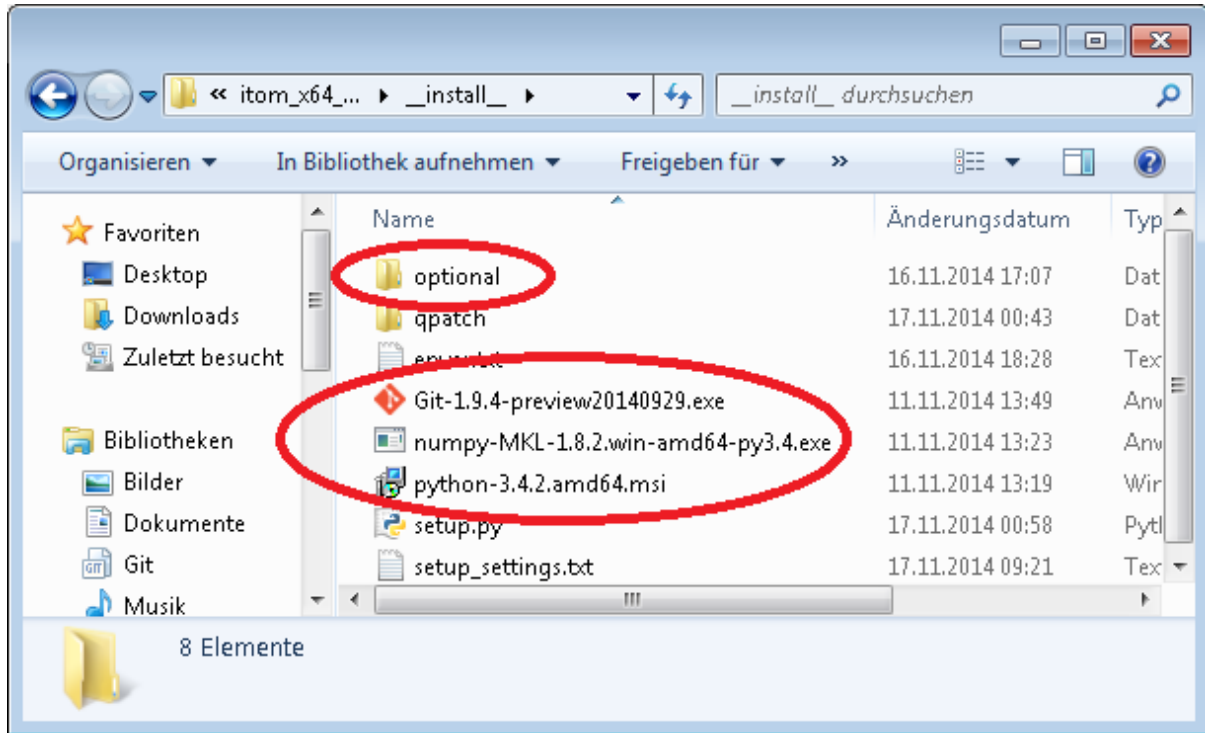
1. Download and unpack one or both archives into the same folder. You should then have the following folder structure:



- `__install__`
- 3rdParty
- 3rdPartyPCL (if you unpack the `..._PCLAddOn.zip` as well)

The `__install__` folder is only necessary during the installation and can be deleted afterwards. After the installation, this folder will also contain a folder **sources** and **build** with the source repositories of itom and their builds.

2. Go to the `__install__` folder



If not yet available, install

- Git 1.9.4
- Python 3.4.2
- Numpy 1.8.2

on your computer. This is required for the further installation. Optionally you can install the **Qt-AddIn** for Visual Studio as well as **TortoiseGit** from the **optional** folder. This can also be done later.

3. Execute the script `setup.py`

The next step is to execute the python script `setup.py` in the `__install__` subfolder. Usually, this should work by double-clicking on the file. However you need to make sure that the file is executed with the Python 3.4.2 version that you recently installed. If this is not the case, make a right click on the file and select **Open with...** and choose **python.exe** from the Python installation path (e.g. `C:/python34`).

This script leads you through the remaining installation process using a menu guided approach:

```

C:\Windows\py.exe
Select the step you want to execute:
-----
1. (OK) clone git directories
2. (??) patch Qt in 3rdParty folder (absolute file names)
3. (OK) configure and generate CMake (itom only)
4. (OK) compile itom in Debug and Release (necessary for further steps)
5. (OK) configure and generate CMake (plugins and designer plugins)
6. (??) compile plugins and designer plugins in Debug and Release
7. execute steps 1,2,3,4,5,6
8. Prepend Qt, OpenCV and optionally PCL to PATH variable
9. exit
-----
your input?_

```

Type any number (1-9) after the question **your input?** and press return to start the corresponding installation step. You can also execute the first six steps using the overall command number 7. Press 9 to quit the setup. You can continue with other steps by call **setup.py** again. Once you executed one step, an **(OK)** after the number shows you that the step already has been executed. An **(??)** means that no further information about a possible execution can be shown. The single step should normally be executed starting at 1 and ending with 8; however you can also omit single steps. The steps are explained in the following points.

Note: If you have already executed **setup.py** at least once, a file **setup_settings.txt** is created in the **__install__** folder. This contains some settings that you made during the installation. Delete this file if you want to restart the installation without preset settings.

(a) Setup: clone git directories

This command clones the **itom**, **plugins** and **designerPlugins** repositories from <https://bitbucket.org/itom> and creates the folders:

- sources/itom
- sources/plugins
- sources/designerplugins

If you have never executed this step before, the setup tries to guess the path to the application **git.exe** (that you installed before). You can accept the guessed path (type **y** and press return) or you can indicate the absolute path to this executable (e.g. **C:/Program Files (x86)/Git/bin/git.exe**). This setting is stored in the settings file **setup_settings.txt**.

(b) Patch Qt in 3rdParty folder

The folder **3rdParty** contains a prebuild version of Qt (5.3.2, with OpenGL support). No further compilation needs to be done. However this Qt installation needs to be patched in order to correspond to your pathes. Use this setup step to execute the patch. If you are able to start executables like **assistant.exe** or **designer.exe** from the **qtbases/bin** folder of Qt, the patch seemed to work.

Note: How is this Qt prebuild version created?

The Qt version in the prebuild setup is obtained by the source archive of Qt 5.3.2 from qt-projects.org. Using MSVC2010 32bit or 64bit, these sources have been configured using the same steps than indicated in [this link](#). The configured project is then compiled (using jom 1.0.14 for a multi-threaded compilation) and only the relevant files are then copied into the Qt5.3.2 folder of your prebuild itom setup.

Note: One drawback of this prebuild Qt installation is that you cannot directly debug into Qt methods (since the delivered ptb-files are not patched and point to invalid pathes). If you want to have this feature (usually not required for standard programming), you need to compile Qt by yourself into the same folder using the approach given in [this link](#).

(c) Configure and Generate CMake (itom only)

At first, **itom** needs to be configured using CMake such that the folder **build/itom** is generated with an appropriate Visual Studio solution. CMake is directly called from **3rdParty/CMake3.0.2**. (If the configuration should fail, the CMake GUI is opened, you can reconfigure anything, generate the project by yourself and close the GUI. Then, the setup will continue.) Before you go on configuring the plugins and designerplugins, you need to build **itom** in Debug and Release first (in order to create the SDK). That is done in the following step.

(d) Compile **itom** in Debug and Release

This step compiles itom from **build/itom** in a debug and release compilation. Then the executables **qitom.exe** and **qitomd.exe** are generated. If you try to start them, this may fail, since the pathes to the binaries of Qt, OpenCV and optionally the PointCloudLibrary are not included in the Windows path environment variable yet (see step 8)

(e) Configure and generate CMake of plugins and designerplugins

Similar to step 3, the plugins and designerplugins are now configured and generated (folders **build/plugins** and **build/designerplugins**). This step will fail, if the itom-SDK could not be found in build/itom/SDK.

(f) Compile **plugins** and **designerplugins** in Debug and Release

Same than step 4 but for plugins and designerPlugins.

(g) Bundle

In order to execute steps 1-6 manually, you can execute all these steps in one row using command 7 (sometimes you need to accept intermediate steps by pressing return)

(h) Modify Windows path variable

In order to find the binaries of Qt, OpenCV and optionally the PointCloudLibrary, it is necessary to prepend some pathes to the Windows path variable. If you choose option 8, a string is print to the command line and saved in the file **enver.txt**. Copy the string and **prepend** it to the PATH environment variable of Windows. Afterwards it is required to restart the computer or log-off and log-on again.

Now you are done with the setup. If you want, you can delete the entire **__install__** folder.

Try to execute **qitomd.exe** or **qitom.exe**.

3.6 Get this help

The user documentation of **itom** can be distributed in various formats. The main format is called **qthelp**, such that the documentation is displayed in the **Qt**-Assistant that is directly accessible from the **itom** GUI. On windows PCs it is possible to compile the help as a Windows Help Document (chm) or to create latex-files from the help, in order to create a pdf-document. The base format of all these formats is a collection of *html*-pages.

If you compiled **itom** from sources, no compiled documentation file is provided. Therefore, you need to compile the help by yourself:

3.6.1 Build documentation

Necessary tools

In order to be able to build the documentation, you need to have some tools installed on your computer:

1. Doxygen

Doxygen is a source code documentation tool, that parses your C++-source code and extracts the documentation strings for all methods. Additionally, it displays the entire class- and file-structure of your project. **itom**'s user documentation sometimes uses results from Doxygen in order to show relevant C++-structures for programming plugins.

Windows users can download the binaries as setup from <http://www.stack.nl/~dimitri/doxygen/>. Under linux the easiest way is to get the latest doxygen package that is available for your distribution.

2. Python-Package Sphinx

The real user documentation is created in **itom** using a python script that needs the **Python** package **Sphinx** (<http://sphinx-doc.org/>). **Sphinx** itself requires other python packages installed on your computer. For windows users, we therefore suggest to obtain **Sphinx** via the python package tool *pip* or *easy_install*.

For *pip*, download the script *get-pip.py* from <http://www.pip-installer.org/en/latest/installing.html> and save it on your harddrive. Then execute this script with your desired python version either by double-clicking on it, or open a command line and call something like:

```
C:/python32/python.exe get-pip.py
```

Then use the command line again and change to the directory:

```
cd C:/python32/scripts
```

and execute:

```
pip install sphinx
```

or:

```
pip install --upgrade sphinx
```

if you want to upgrade sphinx. All dependent python packages will be installed, too.

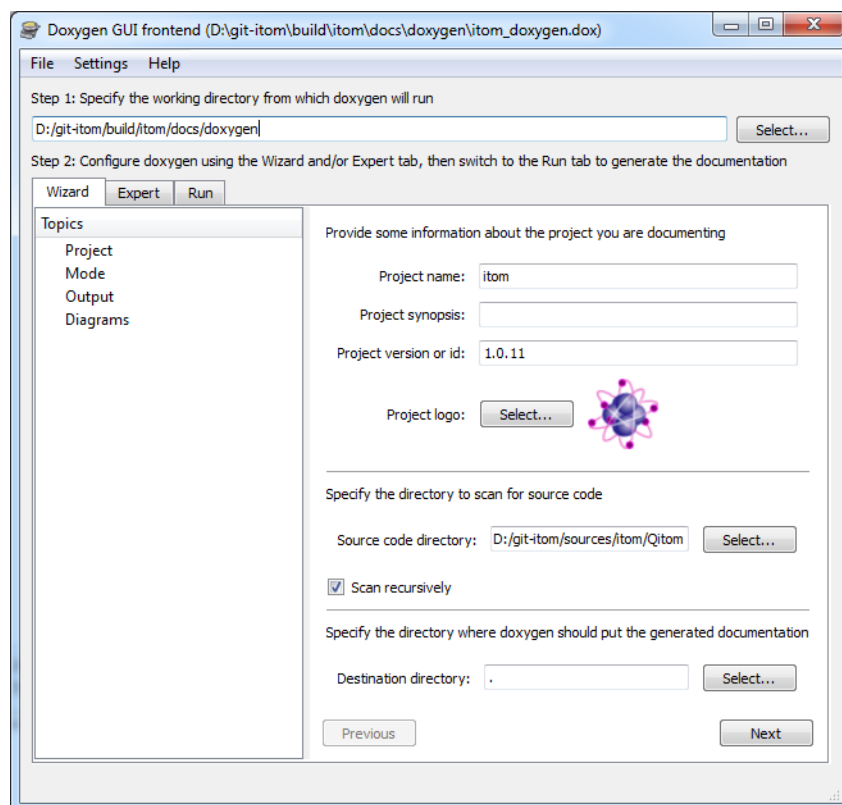
You can also manually download and install Sphinx and its depending packages. Setup-versions of **Sphinx**, **Pygments**, **Jinja2**, **docutils**... are also available from <http://www.lfd.uci.edu/~gohlke/pythonlibs/>. However, you then need to separately install all depending packages of **Sphinx**.

Run doxygen

In your build-directory of **itom**, you will find a folder **docs**. Open its subfolder **doxygen**. There you will find a document **itom_doxygen.dox**. This document contains absolute pathes to the source directories of **itom**'s sources. Run doxygen with this document in order to create the source code documentation.

On Windows computers, the easiest way to do this is open **itom_doxygen.dox** with the tool **doxywizard** that is located in the **bin**-folder of your **doxygen** installation. In **doxywizard** go to the *run*-tab and click on the *run*-button.

After the build process, a folder **xml** is created in the **doxygen** subfolder of the **docs** folder. This **xml** folder is required afterwards.



Run Sphinx

Now open **itom** and execute the script **create_doc.py** in the folder **docs/userDoc** of the build-directory. The default-builder of the documentation is **qthelp**. If you also want to build the documentation for other builders, you can change the list *buildernames*. The following values are possible:

```
qthelp -> default qthelp format for opening the documentation within itom
htmlhelp -> creates a chm-help format on Windows only
latex -> creates a pdf-document using latex. You need to have latex installed on your computer
```

The output of all build processes are located in the folder **docs/userDoc/build/<buildername>**. The locations of the Windows html-help generator or the latex interpreter are detected when running **CMake** for the **itom**-project. The absolute paths to these tools are automatically inserted into the script **create_doc.py**.

Show documentation in itom

When clicking the *help*-button in **itom** or pressing **F1**, Qt's assistant is opened with a set of documentation files. At first, **itom** checks your **itom** installation for various documentation files. Their latest version is then copied into the **help** folder of the build-directory. The search is executed for all **.qch**-files that are located in the **docs/userDoc**-directory.

After having copied the files, a collection-file is generated (containing all qch-files) and displayed in the assistant. If you have a setup version of **itom**, the help-folder already contains a compiled documentation file, that is displayed in this case. Please note that the file check is only execute once per **itom** session. Restart it in order to redo the check.

3.6.2 Plugin documentation

Besides the user documentation, there is also a plugin documentation for all currently available plugins. On a setup installation, the main files for the plugin documentation are contained in the subfolders *doc* of each plugin in the plugin directory of **itom**. Based on these files, **itom** scans their modification date when the **itom** internal help

is called for the first time. If the help needs to be rebuild, a bundle is collected from all plugin sub-documentations and saved in the `docs/pluginDoc/build` directory of the build directory of **itom**. Finally this bundle is packed and prepared for the help assistant of **itom**.

Create the plugin documentation of any plugin

In order to generate the plugin documentation of any plugin, the following requirements need to be fulfilled:

1. In the sources of the plugin there must be a folder **docs** that contains at least on *.rst with the plugin documentation. This documentation needs to be written in the so called reStructured-Text format rst (see <http://sphinx-doc.org/rest.html>)

2. The file **CMakeLists.txt** of the specific plugin must contain the following line in order to register the rst-file as plugin documentation file:

```
PLUGIN_DOCUMENTATION(${target_name} <filenameOfTheRstFile>) #the filename must not contain th
```

3. If the plugin is build, its build folder will get a **docs** subfolder, too. This subfolder consists of a file **plugin_doc_config.cfg**.

If these requirements are given, start **itom** and execute the script **create_plugin_doc.py** in the **docs/pluginDoc** directory of the build directory of **itom**. Then select the *.cfg-file describing the plugin documentation in its specific build folder.

In order to simultaneously create the documentations of many plugins, execute **create_all_plugin_docs.py** and indicate the build folder that contains the build-subfolders of many plugins. These subfolders are searched for appropriate *.cfg files and all sub-documentations are created.

Note: How does this work under the hood?

The most important file for generating the necessary html and QtHelp files from the given rst-files, that are located in the source folder of a plugin, is the configuration file with suffix *cfg* that is located in the build folder of the specific plugin.

This file is generated when CMake executes the CMakeLists.txt file of the plugin and if this file contains the macro **PLUGIN_DOCUMENTATION** like stated above. This macro generates the configuration file based on the template **plugin_doc_config.cfg.in** located in **itom/SDK/docs/pluginDoc**.

The configuration file then consists of the name of the plugin, the located where the source rst-file is located, the plugin build directory where the intermediate files of each single plugin documentation are generated and finally the installation path of each plugin (subfolder plugin of the itom build directory) where important components of the plugin documentation will be copied after having been build.

When executing the script **create_plugin_doc.py** like described above, the plugin documentation (rst-file) is compiled and build in the QtHelp format in the build directory of the plugin. Afterwards the important components are copied into the documentation installation path of the plugin, given by its configuration file.

For creating the final file **itomPluginDoc.qch** in the help subfolder of itom, itom reads all docs subfolder of plugins lying in the plugin subfolder of itom. Then all plugin documentations are collected in the itom subfolder **docs/pluginDoc/build**. From all plugins, the startsite **index.html** is created and the file list, search indices, keywords... from the single plugin qhp-files are merged into the file **itomPluginDoc.qhp**. Finally the Qt Help is created using the qhelpgenerator and qcollectiongenerator tool and copied into the help subfolder of itom.

plugin documentation source files

The source files of each plugin documentation are written in the reStructuredText-format (.rst). *You can use all possibilities given by this format including the additions provided by ****sphinx**** (see <http://sphinx-doc.org/rest.html>). Additionally, when the documentation is build using **itom**, specific directives, roles and a pyitom-domain is added in order to automatically create parts of the documentation depending on information that can for instance be obtained using the command `itom.pluginHelp()`.

Use the following roles as placeholders in the text. The placeholders will be replaced with information obtained by the plugin with the given name. This is only possible if this specific plugin is loaded in **itom**:

```
:pluginsummary:`pluginname` -> short description of the plugin
:pluginintype:`pluginname` -> type of the plugin (DataIO, Actuator, Algorithm)
:pluginlicense:`pluginname` -> license string
:pluginauthor:`pluginname` -> author(s) of the plugin
:pluginversion:`pluginname` -> current version string
```

Furthermore, there are directives that you can use in order insert more information into your documentation file:

For inserting the detailed description of the plugin, write:

```
.. pluginsummaryextended::
    :plugin: pluginname
```

A table with all mandatory and optional parameters that are required to start an instance of a *dataIO* or *actuator* plugin, write:

```
.. plugininitparams::
    :plugin: pluginname
```

The last directive is created for algorithm plugins. An list of all available filters is obtained via

```
.. pluginfilterlist::
    :plugin: pluginname
    :overviewonly:
```

Omit the option `:overviewonly:` in order to get an extended overview of all filters including their mandatory, optional and return arguments. If this overview is inserted all filters in the short list will link to their specific long description.

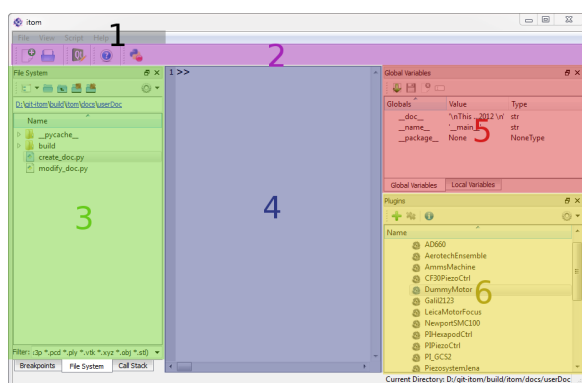
GETTING STARTED

Read the following chapter in order to get a first introduction about the main functionalities of **itom**.

4.1 Quick Start

4.1.1 The GUI

If you open **itom** for the first time, the main graphical user interface will look like the following screenshot:



The main components are highlighted with different colors and numbers. These components are:

1. The menu bar gives you access to the script management, the *script execution and debugging*, the windowing system and help features. You can *add your own menus* to this bar.
2. The most common commands are also accessible via the main toolbar of **itom**. *Add your own toolbars* and connect them with user defined scripts to adapt **itom** to your personal needs.
3. The *file system toolbox* gives you access to script files (written in Python 3) or other files that can be loaded with **itom** (e.g. bmp, png, gif, tif, Matlab files... - depending on the installed plugins)
4. The *console (command line)* allows you directly executing single or multiline script commands.
5. The current set of global (or local) variables in the Python workspace is listed in the *corresponding toolboxes*.
6. All available hardware and software plugins are listed in the *plugin toolbox*. Use this toolbox to connect to your cameras or actuators or get informed about available software filters or algorithms. If you miss a plugin whose library file exists, check the menu **file >> loaded plugins...** so see if the library has been found and / or why this library could not been loaded.

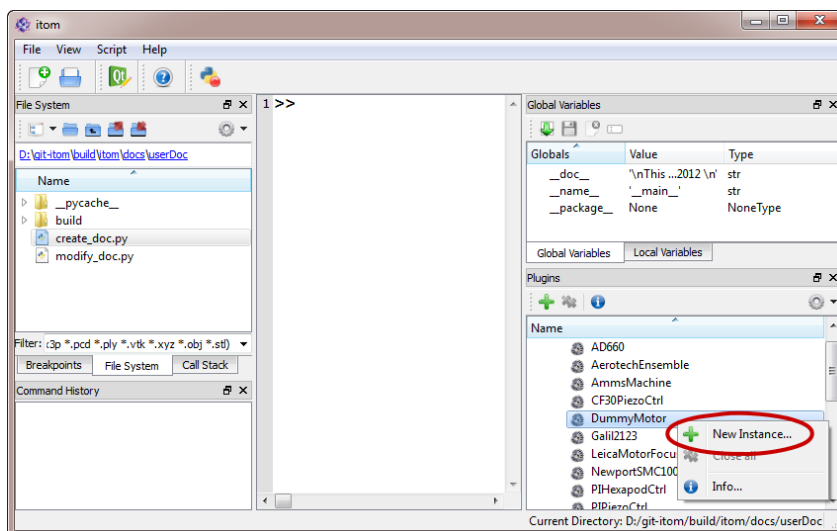
Connected hardware components can be operated either by script commands or directly by the GUI. The following sections show you examples for both approaches.

4.1.2 Connect to a hardware plugin using the GUI

itom comes with a set of different plugins that are used to connect to various cameras, AD-converters or actuators of different manufacturers. You can implement your own plugin (written in C++) in order to make your specific hardware component accessible via **itom**. By default, **itom** is shipped with a virtual actuator and virtual camera in order to simulate hardware on any computer without a real camera being connected.

Note: In this example, an approach is shown how to connect to an actuator instance using the GUI functionalities of **itom**. This approach is very similar for camera plugins.

In the following example, we will connect to such a virtual actuator, move single axes and finally close the actuator. This is all done by the graphical user interface of **itom**. Later we will learn, how to do this by the help of script commands. For this example, you need to have the hardware plugin **DummyMotor** which should be listed in the plugin toolbox as child of the **actuators** group.



Every entry in this toolbox corresponds to one library file (Windows: *.dll, Linux: *.so), located in any subfolder of the **plugins** subfolder of your **itom** installation path. If the library could be loaded, you can connect to one or multiple instances of its represented hardware (e.g. you can connect to various motors of the same type that are connected to the computer). If you don't find the desired library in the *plugins toolbox*, you either don't have the library in the plugins subfolder or the library could not be loaded. This can have various reasons. See the **loaded plugins...** dialog in the menu **file** to check why the plugin could not be loaded.

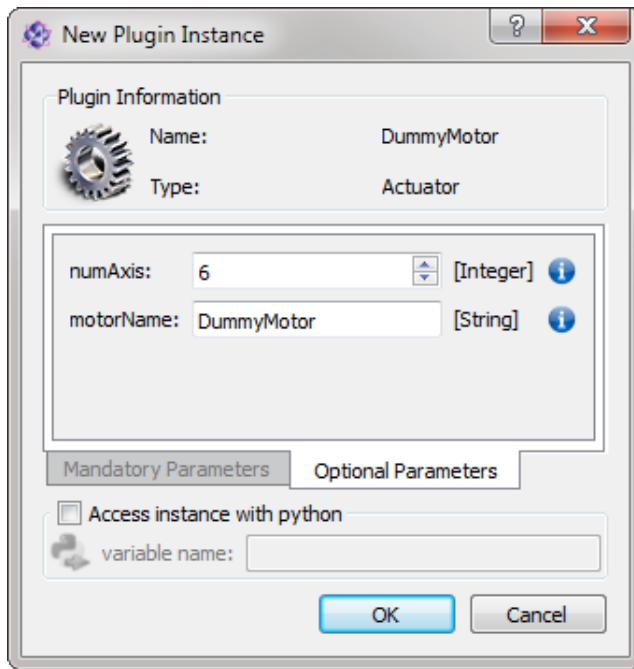
Here, we want to create one instance of the **DummyMotor** plugin which simulates a motor stage with up to 10 axes. In order to create an instance, choose **New Instance...** from the context menu of the corresponding **DummyMotor** library entry (see figure above).

Usually, you have to pass multiple mandatory or optional parameters when connecting to a hardware instance (e.g. a port number, IP address, color space, identification string...). In order to pass these parameters, a dialog is opened:

This dialog is automatically configured depending on the needs of the specific plugin. The tab **mandatory parameters** lists all parameters that you have to pass, the tab **optional parameters** all optional ones. If one tab is disabled, no parameters are required in this section. In case of the dummy motor, no mandatory parameters are required. However, you may change the number of virtual axes or set the internal name of the motor instance.

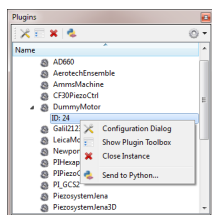
Each parameter has already a default value and a description, that is available when moving the mouse over the information icon at the end of the line. Additionally, parameters can have meta information that may restrict the allowed value range or the string values that are accepted. These restrictions are available via the tooltip text of the corresponding input field.

Press OK to accept the dialog and create the motor instance. If no error message occurs during initialisation a child entry is added to the **DummyMotor** item in the tree view of the plugin toolbox. This indicates that an instance of



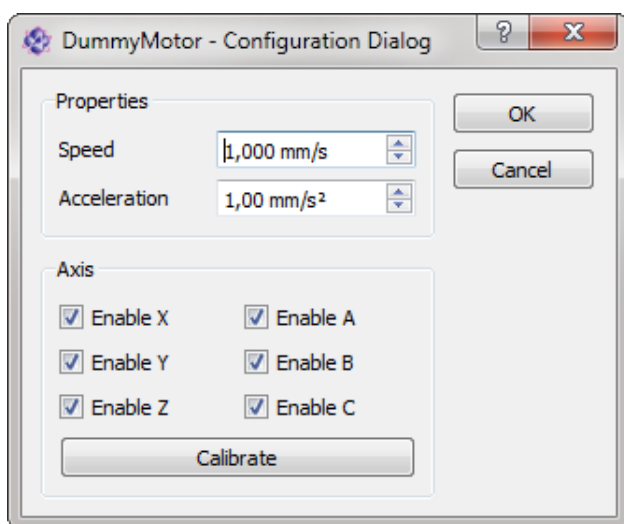
the plugin has been created and is ready for use. The name of this child either corresponds to the optional name of the motor instance or an auto-incremented ID if no name has been given.

Now, you can work with this motor instance. The necessary functions are either available by the context menu of the item or by the toolbar of the plugin toolbox that is always related to the currently selected item (therefore click on the instance's item to get the necessary toolbar):



- **Configuration Dialog:** This opens a modal configuration dialog to configure the actuator or camera instance.
- **Show Plugin Toolbox:** Every plugin can provide a toolbox that is integrated in the main window of **itom** and usually provides commonly used features of the plugin (e.g. the motor movement)
- Click **Close instance** to close the plugin's instance again.
- **Send to python:** If you want to access the plugin's instance by a Python script after having connected it via the GUI, use this command.

The individual (but optional) **Configuration Dialog** of every plugin allows changing relevant parameters of the plugin. Open this modal dialog by clicking the corresponding menu button:



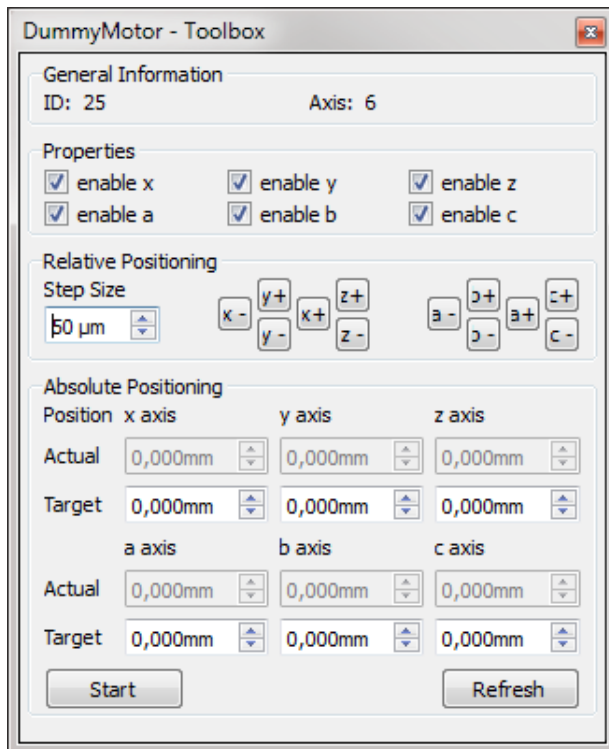
As long as this window is opened, **itom** is blocked for further user input. The configuration dialog is assumed to only contain parameters or functions that are rarely used.

In the next step, we want to move the virtual stage. For this purpose, plugins can provide their specific **toolbox**, that is integrated as dockable widget into the main window of **itom**. Click **Show Plugin Toolbox** to open the toolbox of the **DummyMotor**:

Toolboxes are always non-blocking items that can be arbitrarily placed in the main window and usually contain often used methods and functions. In case of the dummy motor, you can use the toolbox to relatively or absolutely move one or multiple axes. During the movement, related axes are marked with a yellow background color. In case of an error, the background turns to red. The dummy motor instance simulates the movements using a speed value, that is adjustable via the configuration dialog.

Once you are done with the plugin, close the connection to the instance by clicking the **Close instance** button in the context menu or toolbar.

Note: Once you opened the instance by the GUI approach, shown in the recent section, you also need to close the instance by the GUI. The entry in the plugin toolbox is then marked with a gray background. Later, you will see how you can connect to the instance by a Python script or how you can later grant access to the instance to Python scripts. Then the background color turns to yellow. This means, that you have to delete all Python script variables that point to the hardware instance. Once this is done, the instance is automatically closed. For more information, read the help about the *plugin toolbox*.



4.1.3 Scripting

In the next section of the **getting started** we will learn how to execute our first lines of Python code. Python is an open-source and commonly used script language which is fully embedded into **itom**. You can therefore use **itom** to execute arbitrary Python commands or scripts, but you can also use Python to control **itom** and its plugins.

Hello World!

You can get the famous “hello world!” in a very simple way. Just write in the console:

```
print("hello world!")
```

The command is executed immediately and prints the text *hello world!* to the command line. Let us proceed to a more advanced example involving a camera.

Get a camera snapshot

In this example we will explain step by step how to open a camera and make a snapshot. Python code can be written and executed in two different ways, either directly, line by line, using the console, or in the form of complete scripts. In this example we will use the console and will directly type our commands. Later you can write your code in executable scripts or functions and you can control these functions with your own GUI elements.

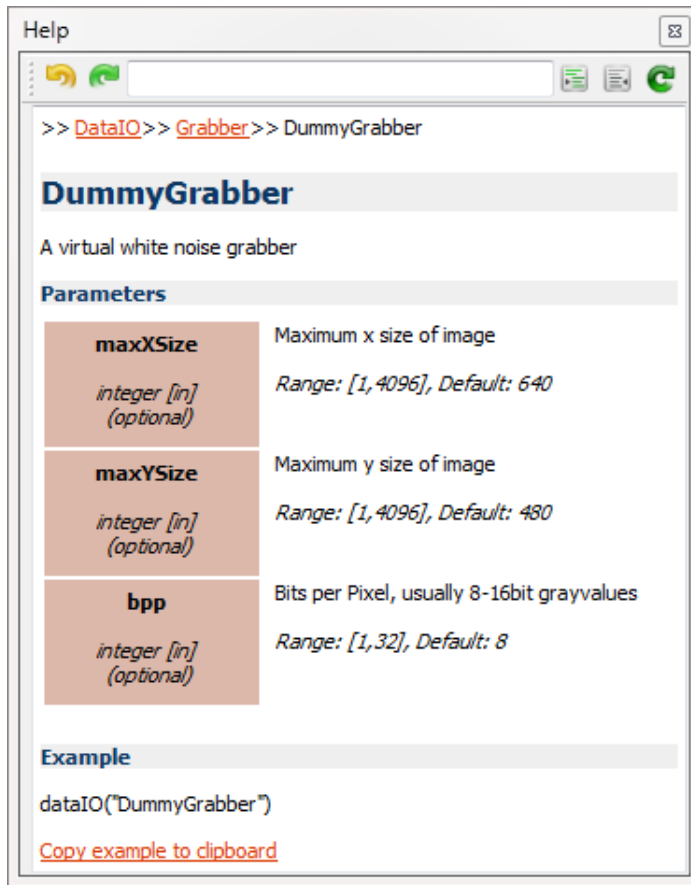
You will need the **DummyGrabber**-plugin for this script to work. This plugin is a virtual camera plugin that provides noise images in order to simulate a real camera.

Step 1: Open the camera

First you have to start your camera device using the respective *dataIO* plugin. In our example we use the plugin *DummyGrabber*, but in general you can use any camera connected to your computer for which you have an **itom** plugin. Similarly to the previous example of the **DummyMotor** we need to create an instance of the **DummyGrabber**. However, we will create this instance using Python script commands instead:

```
camera = dataIO("DummyGrabber", 1024, 800)
```

How did we know what to write there? Select **Info...** from the context menu of *DummyGrabber* in order to show the help page about the *DummyGrabber* plugin:



Note: If you don't see the help page, go to the properties dialog of **itom** (menu **File >> Properties**) and select the checkbox **Show DataIO and Actuator** in the tab **General >> Help Viewer**.

In the help page, all mandatory and optional parameters are listed and described that are needed to instantiate one camera (the same holds for motors). If you remember the connection to the motor above, these parameters have been added to the dialog for the motor's initialization. In case of the camera, there are up to three optional parameters. The first two describe the virtual chip size of the camera, the third one is the desired bit depth, whose default is 8 bit.

In order to create an instance of the camera, we create an object of the Python class `dataIO`, which is part of the module `itom`. Since this module has been globally imported at startup of **itom** using:

```
from itom import *
```

it is allowed to directly write **dataIO** instead of **itom.dataIO**. The constructor of this class requires the name of the plugin (as string) as first parameter. The following parameters are all mandatory parameters followed by the optional ones. You don't need to indicate all optional ones. Therefore, the example only set the parameters `maxXSize` and `maxYSize`. The bitdepth `bpp` was omitted, hence, its default value is assumed.

After initialization, one instance of the camera has been created and is listed in the plugin's toolbox (with a yellow background, saying that it has been created via Python). Additionally, you will find the variable **camera** in the global workspace toolbox of **itom**. Use this variable in order to work with the camera.

Step 2: Get the list of parameters

Every hardware plugin (dataIO or actuator) has a set of parameters. You can read the current value, the allowed range or values and a description of each parameter using Python.

With the following command you get a list with all available parameters for your device.

```
1 print(camera.getParamList())
```

A full list of all parameters including their current value and description is obtained via:

```
1 camera.getParamListInfo()
```

In the printed list you can also see if the parameter is read-only (r) or can also be set (rw) like shown in the next step.

Step 3: Setting Parameters

Parameters of the device that are not marked as read-only can be set as you need it. For example, we might wish to set the upper left corner of the ROI, the integration time (in seconds) of the capturing process or the bit-depth of the captured signal:

```
1 camera.setParam("x0",0)
2 camera.setParam("integration_time",0.05)
3 camera.setParam("bpp",8)
```

Step 4: Getting Parameters

If you are interested in the current value of a parameter you can use the following code to retrieve it.

```
1 sizeX = camera.getParam("sizeX")
2 print("DummyGrabber width: " + str(sizeX))
```

Step 5: Taking a snapshot

Before you can use the camera you have to start it (this must be done only once).

```
1 camera.startDevice()
```

Afterwards, acquire a new image. This is the time when the acquisition starts.

```
1 camera.acquire()
```

If you then want to store the image you have to create a new empty dataObject (the dataObject will resize to the correct dimensions later) and save the values of the image in this dataObject.

```
1 data = dataObject()
2 camera.getVal(data)
```

Here you have to be careful, because data is just a reference (**shallow copy!**, no actual data is copied from the camera's memory) to the internal camera memory. If you want to have a deep copy of the image you should use the copy command.

```
1 dataCopy = data.copy()
```

Alternatively, you can also use the command `copyVal()` to directly get a copy of the object from the camera:

```
1 camera.copyVal(data)
```

Step 6: Displaying the image

Up to now you have not seen any image. You can plot your acquired image using

```
1 plot(data)
```

or you can watch the live image of the camera using

```
1 liveImage(camera)
```

Step 7: Stop/delete the camera

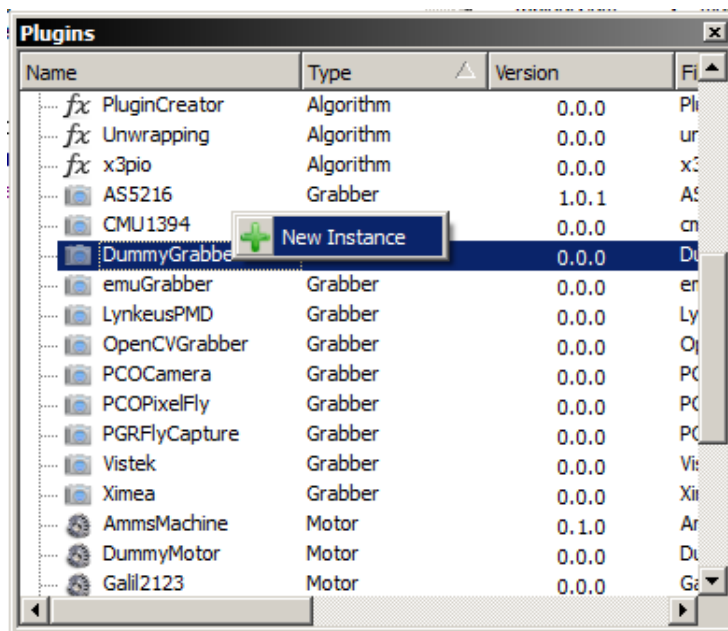
At the end you should stop the camera device. And if you don't need it any more you can delete the instance of the camera plug-in.

```
1 camera.stopDevice()  
2 del camera
```

Alternative Step 1: Open the camera

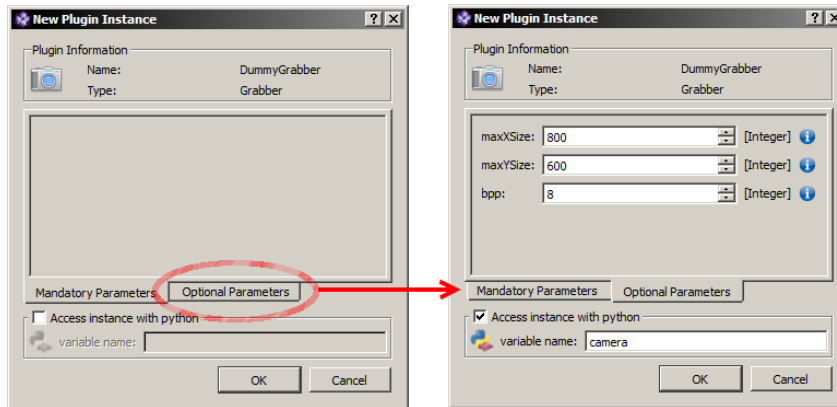
We can also open the camera via the GUI and make the new instance available for access from python.

Therefore we select the **DummyGrabber** within the **Plugins-Browser** (GUI No. 6). We right-click and select 'newInstance'.



The window with initialisation parameters of the plugIn opens. Select the tab **optional parameters** and insert

- maxXSize = 800
- maxYSize = 600
- bpp = 8



Then check **Access instance with python** and type 'camera' into the field **variable name**. Press ok. Now you can proceed with step 2 but since you already set the grabber parameters you can also proceed with step 4.

Apply a median-filter to a snap shot

In the next step, we want to explain how to use filters provided as itom-plugins on the dataObject using the example of a median filter. Instead of executing single python commands from the console, we will now utilize the itom script editor for the first time.

For this example, you will need the **OpenCV-filters-Plugin** and the **DummyGrabber-Plugin**.

Step 1: Open a script editor window

First we open the script editor by left-clicking the **new Script**-button in the button-bar or in the menufilenew Script.



Step 2: Write a simple snap routine

We insert the following code into the editor window. You will recognize the python commands from the previous section.

```

1 camera = dataIO("DummyGrabber")
2 camera.startDevice()
3 camera.acquire()
4
5 dataSnap = dataObject()
6
7 camera.getVal(dataSnap)
8 camera.stopDevice()
9 ...
10 ...

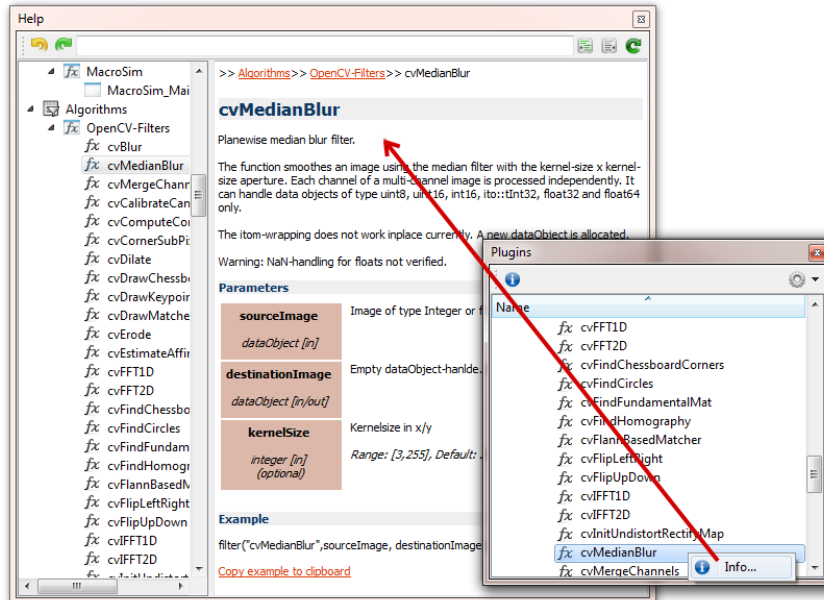
```

Step 3: Retrieve the call for the median filter

What we need in the next step is the correct call for the median-filter, which is defined within the **OpenCV-Filters** plugin. Each plugin provides information concerning its usage that is accessible from python. Therefore we switch to the itom-console and type in

```
filterHelp("cvMedianBlur")
```

You can see a detailed description of the filter “cvMedianBlur”. An alternative approach is to right click on the filter name in the plugin toolbox and to choose **Info...**. Then, the **itom** help widget is automatically opened with the help about the filter as current page:



This help gives you information about the purpose of the filter and its mandatory and optional parameters that are required to apply the filter. At the bottom of the help page, you see an exemplary Python string that shows how to call the filter (only mandatory parameters are included in this example). Click **Copy example to clipboard** in order to simply paste the example into your script or the command line.

Step 4: Insert the filter-call into the script

We use this information and append our script in the script editor with

```
1 dataFiltered = dataObject() #Destination image
2
3 filter("cvMedianBlur", dataSnap, dataFiltered, 5)
```

to filter the content dataSnap into the new object dataFiltered with a 5 x 5 filter kernel.

The filter also works inplace. That means we can use the input object as the output object, overwriting the input data. To show how this works we add

```
1 filter("cvMedianBlur", dataSnap, dataSnap, 5)
```

Step 5: Add plot commands

To view the results we add

```
1 plot(dataFiltered)
2 plot(dataSnap)
```

to the script.

Step 5: Run the script

To run the script, we press the run button or “F5” on the keyboard. If we have unsaved changes in our script, we are asked to save it.



You should see two 2D-Plots.

Getting online help

The python script language has a lot of methods and classes and **itom** expands these functionalities even further. To retain an overview, python provides a built-in online help. You can type `help(class or method)`. For example:

```
1 help(dataIO)
```

gives a complete help for the class **itom.dataIO**, which is the python-representation for various data communication plugIns (e.g. the dummyGrabber).

By typing

```
1 help(dataIO.getParamListInfo)
```

you get the help for this subfunction of dataIO.

To get an overview over the different itom-plugin functions you can use one of several options.

For Plugin-Initialisation, e.g. the DummyGrabber from the last example, use **pluginHelp("pluginName")**.

```
1 pluginHelp("DummyGrabber")
```

For information concerning an already initialized plugIns, e.g. the camera / DummyGrabber from the last example use the member function **.getParamListInfo()**

```
1 camera = dataIO("DummyGrabber")
2 camera.getParamListInfo()
```

To get a list of available (installed and loaded but not necessary instantiated) itom plug-ins of the type filter, use the method **filterHelp()**

```
1 filterHelp()
```

If you want detailed information about one **itomfilter** use `filterHelp("filterName")`

```
1 filterHelp("lowFilter")
```

For user-defined GUI-Elements from plugIns use the function `widgetHelp()`.

4.2 Further Information

For additional information on the features and usage of the **itom** GUI, see GUI *The itom User Interface* For a short introduction to the Python scripting language and use of the **itom** internal dataObject, refer to: *Python scripting language* with more information about the dataObject *DataObject*

4.3 Tutorials

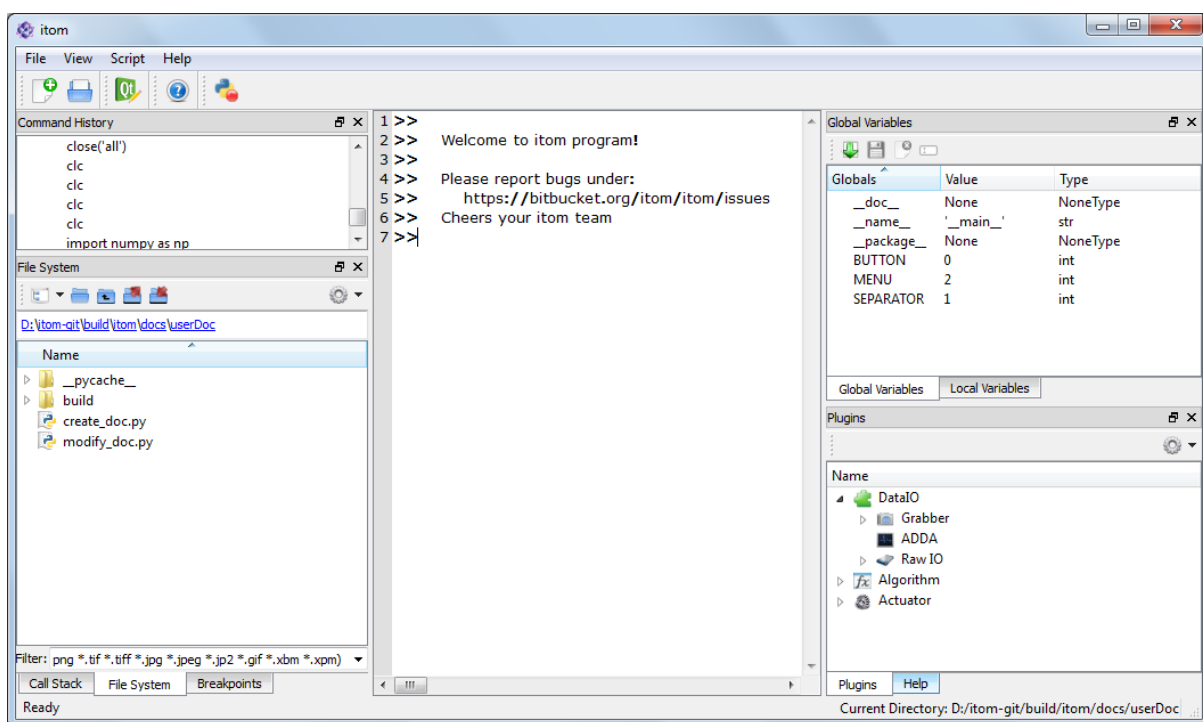
You can find all scripts used here in the **demo** folder. In the *Demo scripts* section of this manual you can find the overview of all demoscripts which are located in this folder and a short summary what is covered in these scripts.

THE ITOM USER INTERFACE

This chapter describes the main functionalities, windows and dialogs that are provided by the main application of **itom**.

5.1 Main Window

During the startup of **itom** a splash screen shows the current state of the load process. After having finished, the **itom** main window appears like in the following screenshot:



The appearance of the application after its start may vary with respect to the depicted screenshot, since the current position, size and visibility of many toolboxes and other components of the overall graphical user interface as stored in the settings at shutdown. They are reloaded at the next startup.

In general, the GUI consists of a command line widget (*console*) in the center and several *toolboxes* that can be arbitrarily positioned as well as moved out of the main window itself (floated state). The toolboxes provide access to many main functionalities of **itom**. Additionally, every opened hardware plugin may provide its own toolbox, that can also be docked into **itom**'s main window.

Further functionalities of **itom** are reached by the menu or the toolbars. It is possible to add further user defined menus and toolbars using the python scripting language (see **toolbar-start_**).

5.1.1 Command Line

The command line in the center of the main window allows executing single or multi-line python commands. Additionally all messages, warnings and errors coming from python method calls or **itom** itself are printed in the command line widget. Errors are highlighted with a red background:

```
Traceback (most recent call last):  
  File "<string>", line 1, in <module>  
RuntimeError
```

Usually, the last line of the command line shows the ">>" sign, that indicates that the console is ready for a new input. You can either write a single python command and press the return key in order to execute it or you can write multiline commands. In order to create a new line for this, press the Shift + Return (smooth line break). After the final command simply press the return key such that the whole command block is executed.

```
>>import time  
time.sleep(2)  
print("I'm awake")
```

The current line or code-block that is executed is highlighted with a yellow background. For multi-line commands, **itom** parses the whole command block and divides it into logical blocks, such that the highlighted background switches from one block to the other one.

In the command line you can use every python command, only character inputs are not supported. For inputs consider using the input dialog mechanism of **itom** (see [msgInputBoxes_](#)). Additionally you can use one of the following key-words in order to clear the command line:




```
clc  
clear
```

To clear the command line from an ordinary script, use the command `itom.clc()`.

Instead of typing all commands in the console, write your entire python scripts in the [Script-Editor](#).

5.1.2 Main menus and toolbars

This is an overview about the menu structure of itom

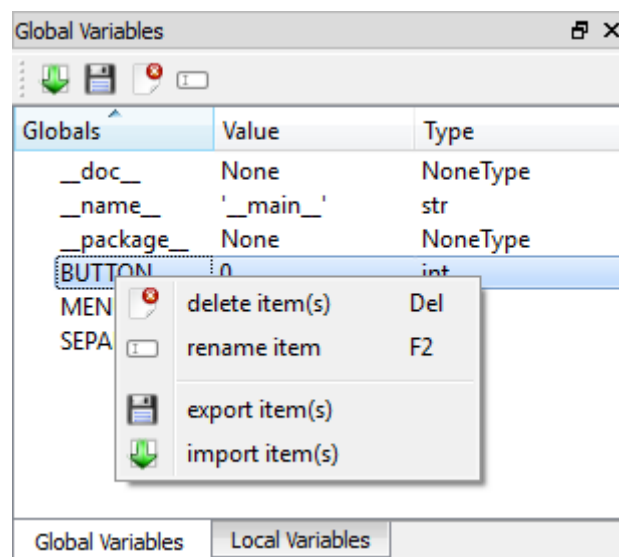
Struc- ture			
File	New Script	 Ctrl+N	Opens a new Python script in the Script-Editor
	Open File	 Ctrl+O	Opens a saved Python script in the Script-Editor
	Properties...		property dialog for important settings of itom
	User Management...		organizing the user dependent appearance of itom
	Loaded plugins...		opens dialog to see the load status of all plugins
	Exit		terminates and quits itom
View	Toolboxes		In the submenu you can toggle the visibility of all toolbars and -boxes
Script	stop	Shift+F10	Stop currently running python code
	continue	F6	Run till end or next break point
	Step	F11	Step into call or if not possible make a stepover
	Step over	F10	Step to next call in the current function
	Step out	Shift+F11	Go on with code till next method call outside current function
	Run ... debug		If toggled python methods triggered by user interfaces is executed in debug.

All the actions mentioned in the table above are accessible by either the menu of **itom** or some of them are also available in some main toolbars that come with **itom**.

5.1.3 Toolboxes

Content:

Global and Local Workspace



In Python, every loaded package has a global workspace, where all global variables are saved. The main workspace of **itom** is a huge dictionary, containing all variables as well as method and function pointers and loaded packages and modules. The content of this global workspace is listed in the toolbox **Global Variables**. There is a filter,

such that all variables of type **method**, **function** and **type** (classes...) are not listed in that overview. If you want to access items of this global workspace you can directly access them by their variable name, e.g.:

```
>>print(__name__)
'__main__' #answer
```

However, this is only possible if you work in the command line or a script that is not part of a separate module. In this case, you can access items of **itom**'s global workspace by importing the module `__main__`:





```
import __main__
print(__main__.__dict__["__name__"])
```


Note: In Python, every module or package has its own global workspace. The main global workspace depicted in the toolbox is related to the global workspace of **itom** that is always accessible by other modules using the dictionary `__dict__` of the `__main__`-module.

The toolbox **local variables** is only enabled if you are currently debugging a python script and the execution is stopped at a breakpoint or in a certain line. Then, the current workspace (local scope) of the method, where the debugger has been stopped, is shown. If the debugger currently processes any code not contained in a method, no local workspace is available.

Double-clicking on an item opens a small dialog where the content of the variable is printed. This is the same result as:


```
print(VARIABLENAME)
```

The context menu, depicted in the figure above, shows some important methods of this toolbox. Most of them are also accessible by the toolboxes toolbar. You can always delete  one or multiple selected variables or you can rename  one selected variable. Additionally, there is the possibility to import  or export  variables to or from this workspace.

A click on **import**  opens a file dialog where you can choose a specific file to import. The following file formats are loadable:

- itom data collection (.idc). This is a pickled file, that may contain several python variables and is loaded using the module **pickle** of python.
- Matlab file (.mat). You can load matlab files in **itom** if you have the Python package **scipy** installed.
- Several itom algorithm-plugins contain methods that implement one of the following interfaces: **iReadDataObject**, **iReadPointCloud**, **iReadPolygonMesh**. Then the data formats that these methods can load are accessible in the dialog as well for loading.

Note: Existing variable names will be replaced unless the variable points to a class, method or function.

If you selected one or multiple variables and click on **export** , another file save dialog is shown where you can choose a specific filename in order to export the variables. You can export into the following file formats:

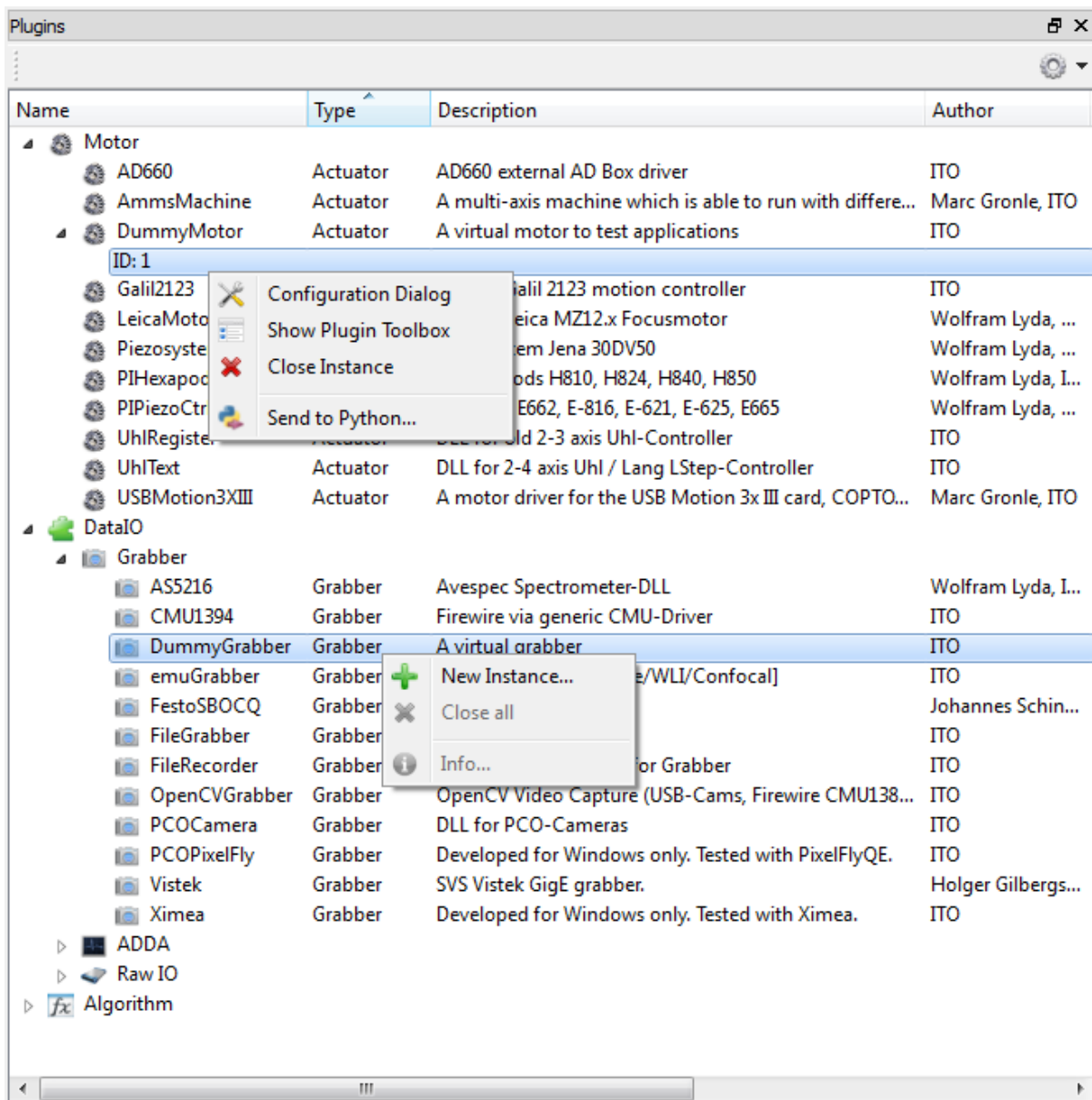
- itom data collection (.idc) [multiple files]. The variables are pickled (python module **pickle**) to the file.
- Matlab file (.mat). [multiple files, Python package **scipy** required].
- If you have itom algorithm-plugins installed that implement one of the following interfaces: **iWriteDataObject**, **iWritePointCloud**, **iWritePolygonMesh**, then you can also export dataObjects, pointClouds or polygonMeshes into the specific file formats. [single variable only].

In both the file open and save dialogs, the filter list always considers all file formats that are available for import and export on your computer.

All variables which are defined in python console or in any python script are stored as global variables. These can be seen in the Global Variables Toolbox.

In order to insert the name of a variable into the *command line* or a *script editor widget* you can drag-and-drop a root-level variable into the destination widget. Currently, the drag-and-drop is only available for root-level variables. In any other cases you can access the entire string to the variable or its subitems (for lists, tuples, dictionaries, classes...) by double-clicking on the item. Then, a dialog is opened that shows details about the clicked item. Click the copy-icon next to the name of the variable in order to copy the name string into the clipboard.

Plugins



The plugin toolbox shows all loaded plugins included any opened instances. The plugins are divided into the following types:

- Actuator (Motor) -> any actuator plugins.
- DataIO >> Grabber -> any cameras.
- DataIO >> ADDA -> any analog-to-digital or digital-to-analog converters.
- DataIO >> RawIO -> further input/output plugins.
- Algorithm -> all algorithm and extended-widget plugins.

The plugins of type **actuator** or **dataIO** behave in the way, that it is possible to open one or multiple instances of every plugin, while all algorithm plugins always provide a set of multiple algorithms (filters) or user interfaces.

At startup of **itom** all plugin libraries, located somewhere in the subfolder **plugins** of **itom** are loaded. If the load fails (for instance due to missing 3rd party libraries), their load is aborted and the reason can be seen in the dialog **loaded plugins...** (menu **file >> loaded plugins**). A right click to any hardware plugin opens the second context menu, depicted in the figure above. There you have the possibility to open a **new instance** of this plugin. After having created this instance, the plugin instance is unfolded and you see the instance by its identifier or ID like in it is the case for the plugin **DummyMotor** in the figure above. The context menu of an instance is also depicted in the figure.

Depending on the implementation of the plugin you have the following possibilities:

Name	Description
Configura- tion Dialog	If the plugin provides this a configuration dialog is started (else: disabled)
Show Plugin Toolbox	If provided, the instance's toolbox is shown in the main window (else: disabled)
Close Instance	closes this instance (see note below)
Send to Python...	You can access this plugin's instance by a global variable in Python under the name that you give in the input box that appears after clicking on this action.

Note: If you want to close an instance of a plugin by this toolbox, you need to consider the following remarks:

- If you opened the instance by the toolbox and no python variable is created for this plugin, you can simply close it by the GUI
 - If you created the instance by the GUI, however created a python variable, a click on **close instance** deletes the reference, the GUI is holding to the plugin's instance. However, the instance is only closed if all Python references are deleted as well.
 - If you created the instance by Python, you cannot delete it by the GUI.
-

File System

The file system toolbox gives you access to the file system of your harddrive.

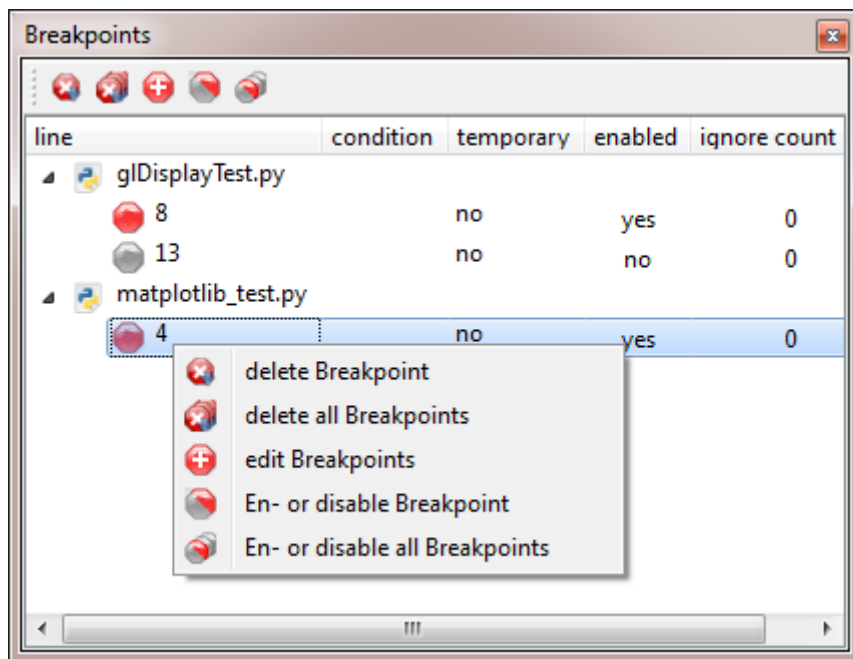
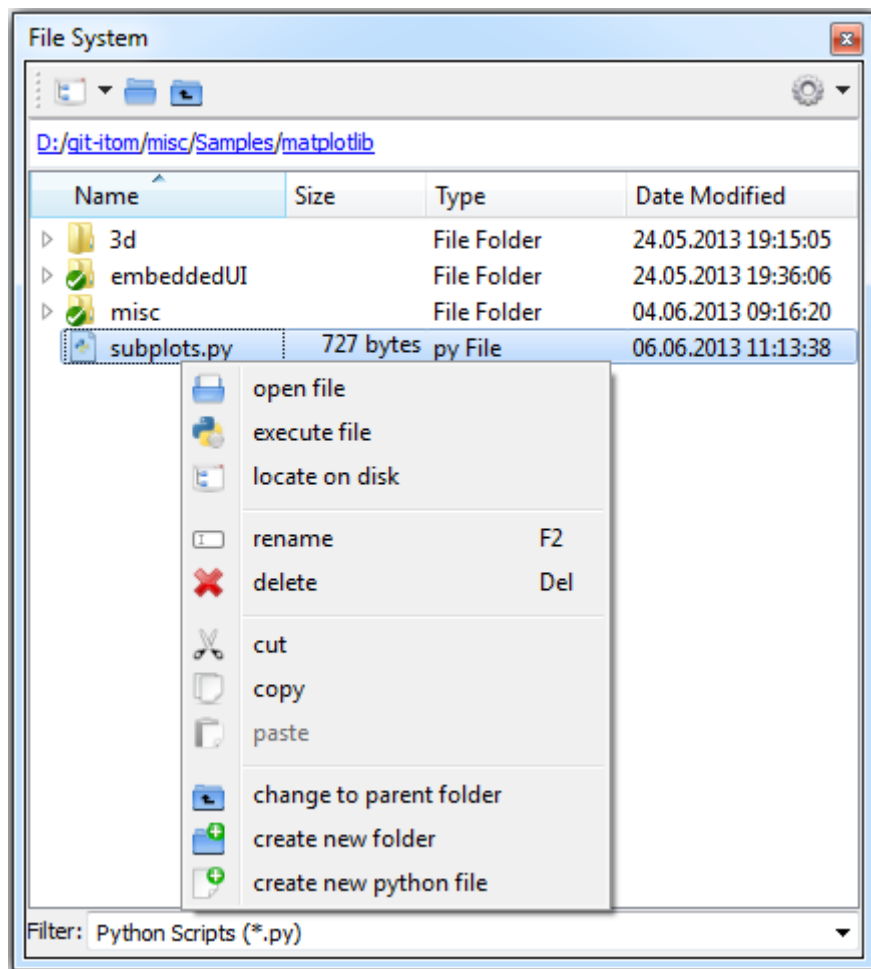
Below the toolbar, a breadcrumb menu allows seeing the current path, where a click to any parent folder changes the current path to that folder. The current path is also printed at the right side of **itom**'s statusbar and also corresponds to the current, active directory of python. The filter below the file system, contains file filters to all file formats that are recognized by **itom**, either by a direct way (Python files, itom data collection, ...) or by algorithms implemented in any algorithm plugins. However you can also type your own filters in order to filter the file system tree.

The first combo box in the toolbar gives you access to the last ten current directories, that are also saved at termination of **itom** and are available at the next startup. The context menu of every item depends on the type of the item (file, folder, empty space in the tree). A right click on a python file for instance allows to open that file or to directly execute the script. The command **locate on disk** opens an external explorer with the current directory as start directory. Additionally, you can always create a new directory or python script in the current folder.

The file system toolbox also provides the common drag&drop functionalities allowing to move or copy files from and to another application, like the explorer. You can even drag a data file, that is known by **itom** onto a workspace toolbox in order to import that file in the workspace.

Breakpoints

As you already learned in the section *debugging python scripts*, it is possible to set and configure breakpoints in all scripts. Once the python interpreter is executed in debug mode, it will stop at any defined breakpoint.



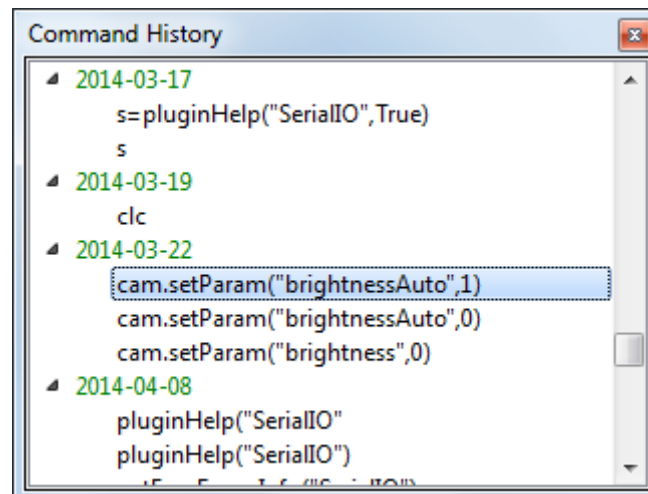
The breakpoint toolbox gives you an overview about all breakpoints that are currently registered in the python debugger. These breakpoints are saved on shutdown of **itom** and restored at the next startup. Breakpoints can even be active if the corresponding script file is currently not opened; it will be opened once the debugger stops at the corresponding breakpoint.

In the toolbox, all breakpoints are sorted by the script file and the line number. A red dot indicates that the breakpoint is active, a gray dot stands for a disabled breakpoint. Using the context menu or the buttons in the toolbox, you have the following possibilities:

- Delete the breakpoint that is currently selected (one single breakpoint, the one that has the focus).
- En- or disable the breakpoint that is currently selected.
- Delete all breakpoints
- Toggle the status (en- or disable) or all breakpoints
- Edit the breakpoint that is currently selected.

The configuration of every breakpoint is displayed in the corresponding columns. For more information about this, please see the [documentation](#).

Command History



If the command history option is enabled in the properties of **itom**, every script command that is executed via the command line, is chronologically saved in the command history toolbox. Depending on the properties, all entries can also be grouped by the day of their execution. There is a limit of saved commands (default: 100), such that older commands are automatically deleted from the toolbox if the maximum number of commands is reached. At shutdown of **itom** the current list of commands is saved and reloaded at the next startup.

In the properties of **itom** there is another option, that forces **itom** not to display multiple equal commands in one sequence. Only the first command is saved then.

To clear the entire history, choose **clear list** from the context menu of the command history.

It is possible to drag&drop one or multiple commands from the history to either any script window or the command line. For selecting multiple commands, use either the shift-key (adjacent commands) or the ctrl-key for selecting single commands.

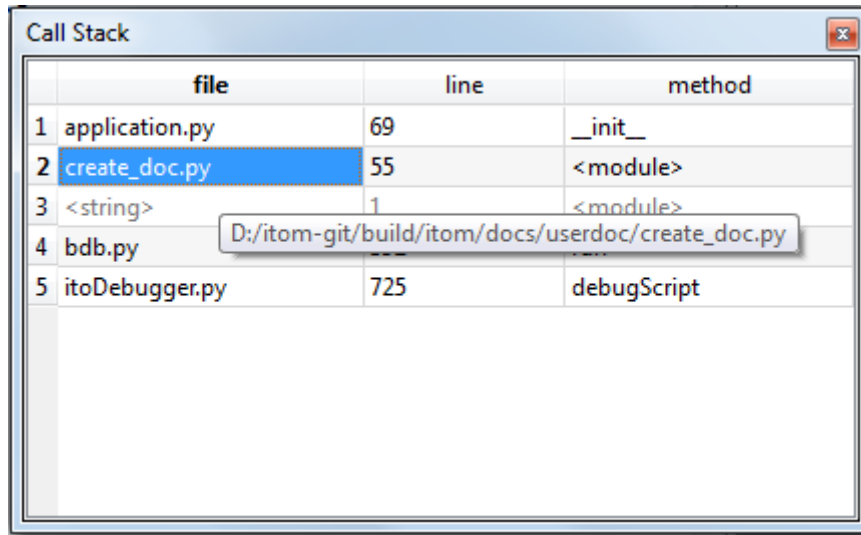
Note: When dropping one or multiple commands onto the command line, they are not directly executed. Therefore press the return key in order to start the execution.

Call Stack

The call stack toolbox is only active and useful when debugging any python script with **itom**.

The usage of the call stack should be presented using one simple example. The script **create_doc.py** is executed in debug mode and the debugger is currently waiting in the constructor of the class **Sphinx**, implemented in the file **application.py**. When opening this file in **itom**, a yellow array shows the current position of the debugger in line 69 of the file **application.py**.

The corresponding call stack is then visible in the toolbox, like depicted below:



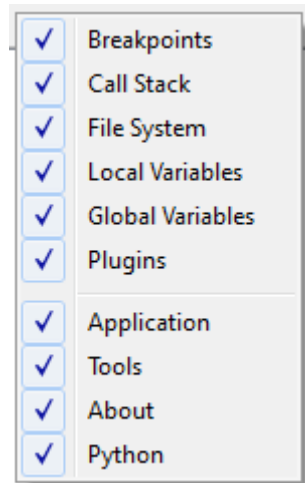
A call stack shows the current function stack. At first, the debug process in **itom** is started by calling some functions in the files **bdb.py**, **itoDebugger.py** and **<script>**. Afterwards the currently debugged line in the script, where the debug process is started, is listed. If this script calls another method or class in the same or a different script, an entry is added on top of the call stack. Once this method is finished, the entry is removed from the stack and the debugger continues at the position, that is currently on top of the stack.

By double-clicking on any enabled entry of the call stack, the corresponding python script is opened in **itom** and the position of the debugger becomes the current line. You can use the call stack in order to trace the entire call history for an improved debugging of the script(s).

Note: If you think, that the debugger shows another line in the script that does not correspond to the real executed code, you may need to reload this or further scripts. Python has an improved caching mechanism for all modules and packages that have been imported at any place of your script. Once imported, the modules and packages are translated in an intermediate file that is cached and saved in a **__pycache__** folder. Further changes in any related script file are not executed if the old files are still cached. In order to force **itom** reloading any modules, either delete the cache folders, use the **reload** method from the module **imp** or use the dialog **reloadModules** from **itom**. For more information about reloading plugins see [here](#).

The core application of **itom** already comes with a set of different toolboxes. Additionally, many plugins provide the possibility to open a toolbox for every opened hardware instance, like actuators or cameras. These toolboxes are then also inserted into the main window of **itom**. Usually toolboxes can be docked into the main window or be in a floating state, such they appear like an unresizable window. If docked, they can be positioned at the left or right side or the top or bottom of the main window. Some of them however are limited with respect to the dockable positions.

All available toolboxes are listed in the menu **View >> Toolboxes**, where hidden toolboxes can be shown again. Additionally, a right click to any place in the toolbar opens the following context menu where the first items also access the loaded toolboxes. The items after the separator correspond to the toolbars, such that they can be hidden or shown:



It is possible to (un)dock the Toolboxes to the main frame at different positions. This is done by simple drag and drop of the titel bar of the toolboxes. Another way of (un)docking can be realized by double-clicking on the title bar.

At the startup of the iTOM software all 5 Toolboxes are activated, which are:

The following main toolbars are available:

- [Global and Local Workspace](#) shows the global and local workspaces of Python.
- [Plugins](#) shows all loaded plugins including opened instances.
- [File System](#) gives you access to the file system of your harddrive.
- [Breakpoints](#) shows all breakpoints added to Python scripts.
- **callstack** shows the callstack when the Python script execution stops at a breakpoint.

5.2 Script-Editor

The script editor is one of the main windows of **itom** and allows creating, modifying and executing one or several python scripts.

Every script editor window can display one or multiple script files that are separated in different tabs at the bottom of the window. Click a tab to show the corresponding script in the center of the window. You can close, save, reorder or (un-)dock every single tab by its context menu.

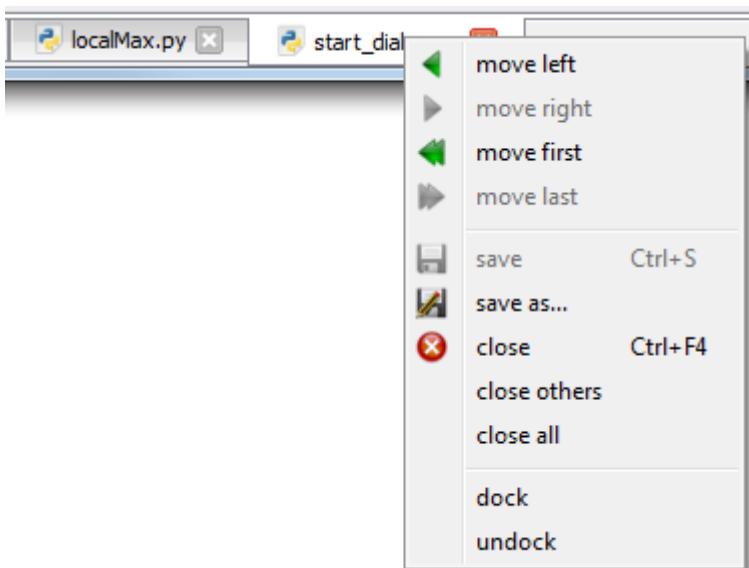
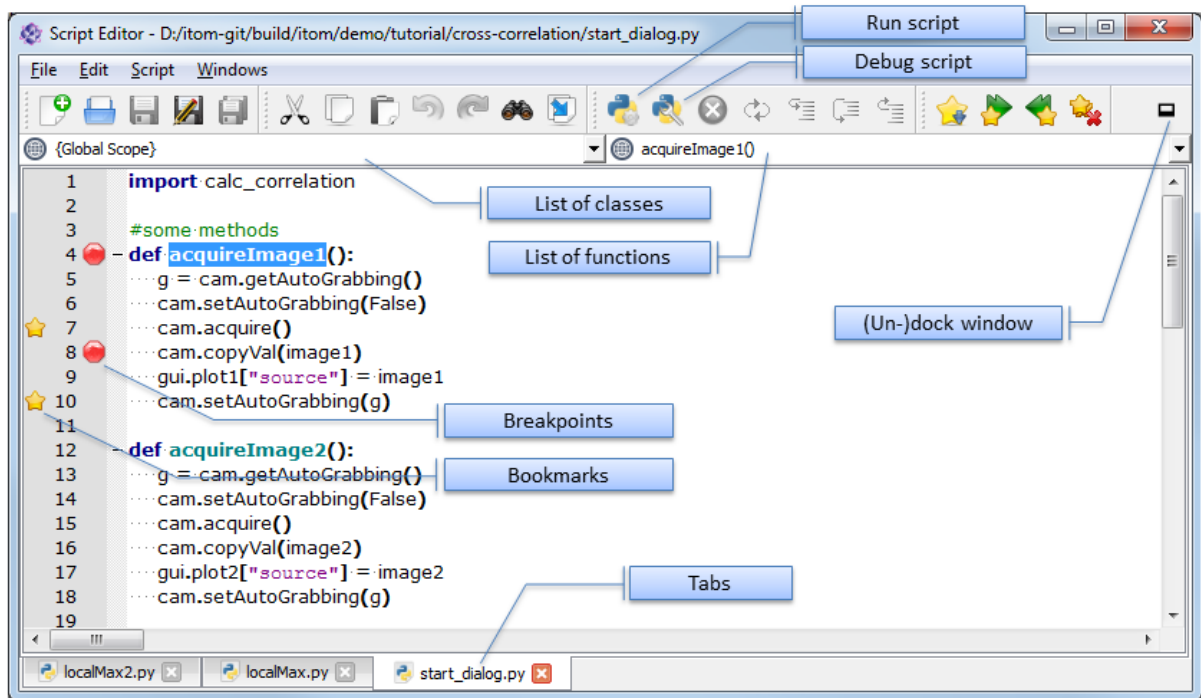
The content of the current script file is displayed in the center of the script editor window. Use this text editor to write or modify a script that can then be run or debugged. The menus of the script editor window provide many possibilities to edit and finally run or debug the script. The most important functions are also accessible via the toolbar and / or the context menu of the script area.

5.2.1 Basic functions

The file menu gives the opportunity to open an existing script or create a new one, to save the current script or to print or close the script.

Basic and advanced functions to edit the current script are contained in the **edit** menu. These are:









- cut, copy and paste parts of the script.
- comment (Ctrl+R) the selected lines using the #-character or uncomment lines (Ctrl+Shift+R).
- indent or unindent the selected lines.



- open a search bar at the bottom of the script to search for a string (quick search).
- open a search and replace dialog.
- open a goto-dialog (Ctrl+G) where you can enter a specific line number the cursor should be moved to.
- open the *icon browser* to search internal icons of itom and loaded plugins that can also be used for specific user interfaces.
- You can set bookmarks by clicking the left margin (left from the line numbers). A star icon indicates a bookmarked line. Modify the bookmarks or jump to the next by clicking the corresponding menu entries.

5.2.2 Run or debug the script

In order to run or debug the script, use the functions given in the **script** menu.

	run	F5
	run selection	
	debug	F6
	stop	Shift+F10
	continue	F6
	step	F11
	step over	F10
	step out	Shift+F11

These are:

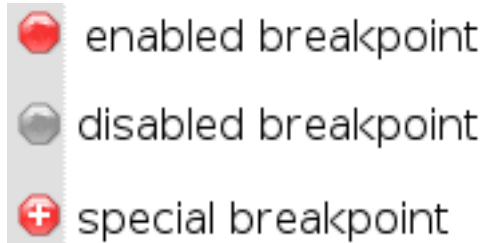
- **Run Script** (F5): Click this to run the current script. This is the default way to execute a script.
- **Run Selection**: If you mark a specific part of the script and choose **run selection**, this selection is copied into the command line and executed. Please notice, that the number of leading spaces of the first selected line is also removed from the following lines.
- **Debug** (F6): Click this to debug the current script such that you can jump from line to line or breakpoint and see the state of the script and all global or local variables. The line where the debugger is currently stopped is marked with a yellow arrow. Then, the debugger waits for your input how to proceed. The options are...
- **Continue** (F6): Continue the script execution until the next valid breakpoint or the end of the script.
- **Step** (F11): The script executes the following line or jumps into the first line of the function that should be executed in the currently marked line.
- **Step Over** (F10): Almost the same than **step** besides that the debugger always executes the script until the next line in the current function or the main script. Further function calls are entirely executed.
- **Step Out** (Shift+F11): Executes the script until the end of the current function and stops in the next line of the caller.
- **Stop** (Shift+F10 or Ctrl+C): Stops a currently running script (run or debug mode). Please notice, that the script can not always be stopped immediately. For instance, the stop flag is not checked when a sleep command from python's time module is executed.

The functions **continue**, **step**, **step over**, **step out** and **stop** are only active if a script is currently debugged or run (stop only). These functions are also accessible via the script menu of the **itom main window**.

More information about breakpoints are given in the next section.

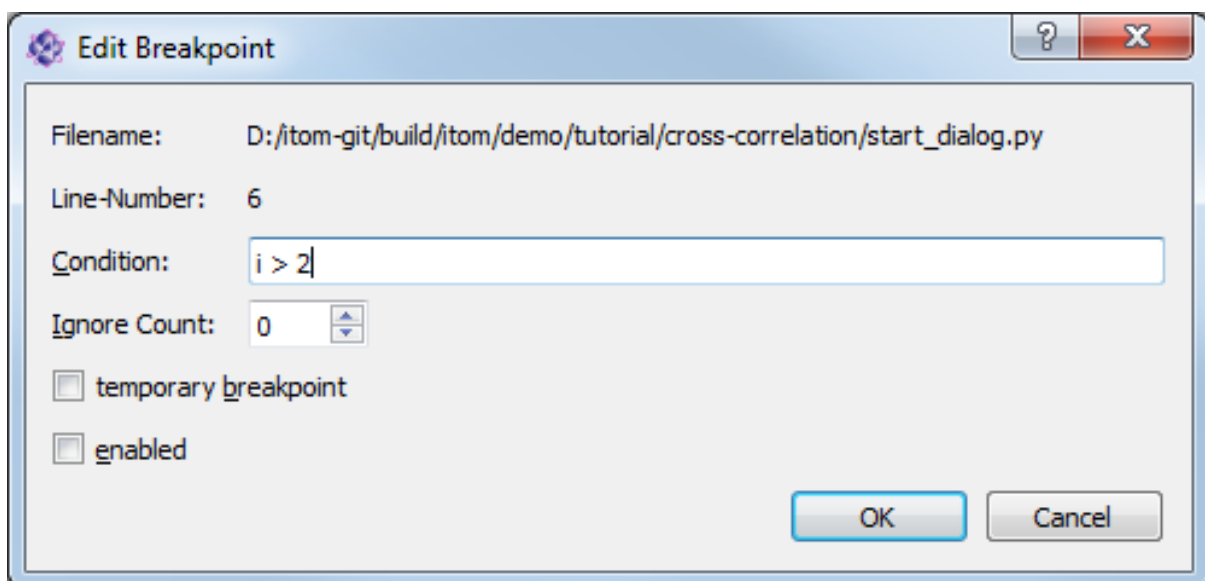
5.2.3 Breakpoints

Breakpoints are red or gray points in the right side of the margin of a script window. In every valid script line, there can be at most one breakpoint. A breakpoint is added to a specific line by clicking the margin at the right side of the line numbers. You cannot add a breakpoint to a commented or empty line. If you clear or comment a line that already contains a breakpoint, the script debugger will fail with a corresponding error message.



There are three different types of breakpoints:

- Red point: This is an enabled, standard breakpoint. Once the script is debugged, it will always stop at this breakpoint.
- Gray point: Disabled breakpoint. This breakpoint is currently inactive. You can enable or disable a breakpoint by clicking it or via its context menu.
- Red point with white cross: Special breakpoint. Right click on a breakpoint and choose **edit breakpoint** to set further settings to the breakpoints behaviour.




The **edit breakpoint** dialog allows configuring the following properties of a breakpoint:

- Condition: Indicate a python statement that is evaluated once the debugger comes to the corresponding line. It will only stop at this line if the condition returns true. You can use any global or active local variables inside of the condition.
- Ignore Count: If this number is bigger than zero, the debugger ignores this breakpoint for the first n times, where n is the value of **ignore count**.
- Temporary Breakpoint: The debugger only stops at this breakpoint once and ignores it after having stopped there for the first time.
- Enabled: En/Disables the breakpoint.

The breakpoints of this and other scripts are all listed in the *breakpoint toolbox* of **itom**. If **itom** is closed, all current active and inactive breakpoints are saved and restored once **itom** is started again. Breakpoints are also active if the corresponding script is currently not visible in any script editor window.

Note: In order to stop the script execution in a debug-mode in any method that is for instance called by clicking a button in an user-defined interface or via a timer event, you need to set a breakpoint in the corresponding line in

the script and toggle the button  **Run Python in debug mode** in the main window of **itom** (toolbar or menu **script**). The same holds for a method that you call from the command line. Toggle this button and set a breakpoint in the method in order to run this method in debug-mode and let the debugger stop in the line marked with the breakpoint.

5.2.4 Syntax highlighting and auto completion

A script highlighting mechanism is implemented to simplify the reading and programming of the script. You can change the styles of the syntax highlighting in the *property dialog* (tab *styles*) of **itom**.

Another big feature is the right handling and help for working with indentations using spaces or tabs. The python language is structured using indentation. Each indentation level always needs to consist of the same amount of spaces or tabs; additionally you must not switch between tabs and spaces for the indentation within your scripts. The script editor has a feature to automatically replace tabs by a certain amount of spaces (it is recommended to set this feature and use four spaces for one tab). Additionally, you can display spaces or tabs and be warned if you switch between both. All these features are configurable in the *tab general* of the property dialog.

When you start typing a new command in the editor window or the command line, it is possible to display either an auto completion list or calltips. The auto completion list appears after having typed a certain amount of characters of a new word and displays similar commands or keywords. You can choose from this list using the tab-key. It is possible to choose the sources for this list:

- Use the recently typed words as sources
- Use words contained in so called API files as sources. The API files can be set in the *tab API* of the property dialog.
- Use both sources

The auto completion list settings are listed in the *tab auto completion* of the property dialog.

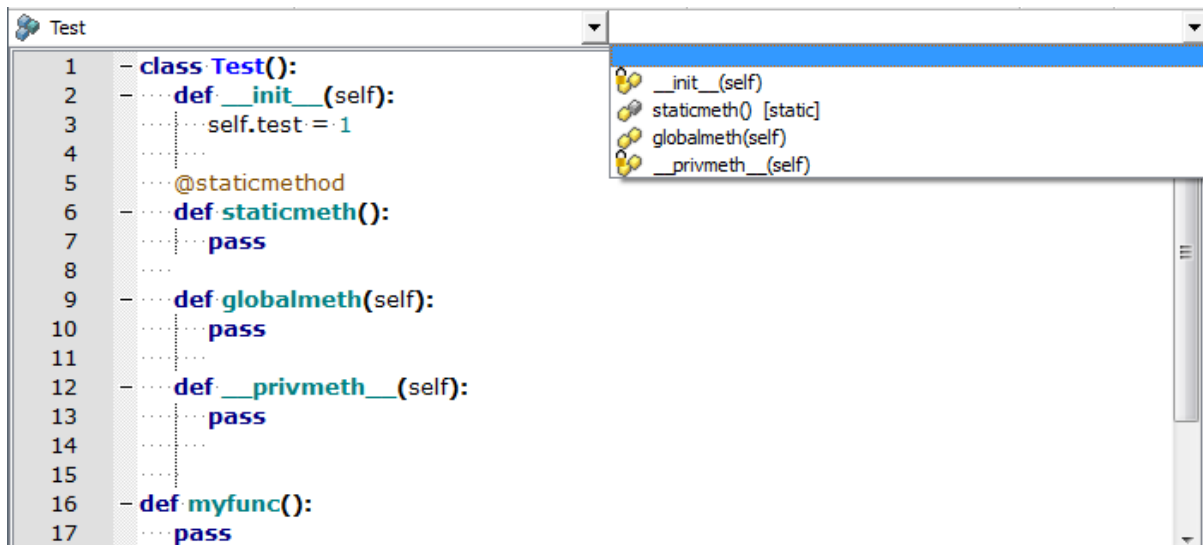
Calltips always appear if you open a rounded bracket to give the arguments of a function call. Then the first line of the docstring of all functions that are listed in any loaded API file and have the same name than the written function are displayed. Configure the calltip behaviour in the *tab calltips* of the property dialog.

5.2.5 Direction class and method navigator

Above every script, there are two combo boxes that are part of a class and method navigator. If these combo boxes are not available, you need to enable this navigator in the property dialog, *tab general*. After a configurable number of seconds after the last change in the script, it is analyzed and the combo boxes are adapted with respect to the current structure of the script.

The left combobox displays all classes and the global namespace of the script. By clicking on any class name, the cursor jumps to the class and the name is highlighted. The right combobox shows the sub-items that belong to the chosen class or namespace.

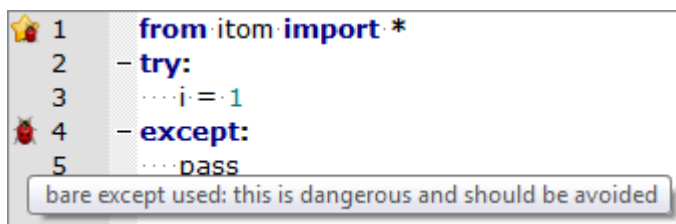
The navigator can distinguish between public and private methods, static methods (with the decorator **@staticmethod**) and specially marked class methods (decorator **@classmethod**). All globally defined methods are categorized into the global namespace (**Global Scope**).



5.2.6 Automatic syntax check

If desired, the current script can be checked for syntax and other errors or hints. This is done using the python package **frosted** (<https://pypi.python.org/pypi/frosted/>). You need to have this package installed in order to benefit from this service. If **frosted** is not installed, the syntax check is automatically disabled.

If **frosted** is installed, the syntax and style check can be dis- or enabled and configured via the *property dialog of itom*. Syntax bugs or other hints, warnings or errors detected by **frosted** will be displayed via a bug symbol in the left margin of the script editor window:



The tooltip text of every bug icon displays the reason for the bug. Since the bug icon is displayed in the same margin column than the bookmarks, there is also a combined icon for a bug and a bookmark in one line. Please note, that one line can also contain multiple syntax hints, they are displayed in multiple lines in the tooltip text.

Note: You can automatically download and install **frosted** using the *python package manager* of **itom** accessible via the *Script* menu of the main window. Click *install* in the manager and search the python package index for the package **frosted**. Try to call:

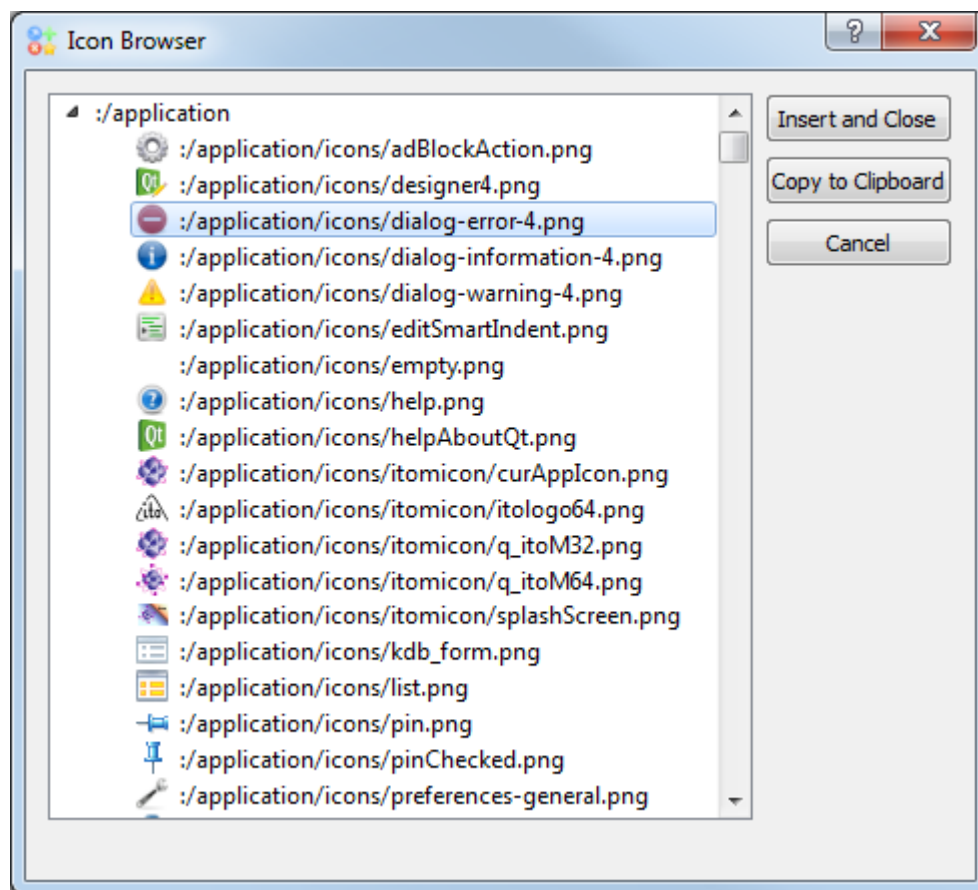
```
import frosted
```

to check if frosted is installed correctly.

5.2.7 Icon-Browser

To help adding icons to user defined buttons, menus or user interfaces, the icon browser of the script editor window shows a list of icons that come with **itom** or loaded plugins. The icon browser is opened via the menu **edit >> icon browser...** of any script editor window or the keyboard shortcut **Ctrl + B**.

These icons can directly be used inside of any script by their *virtual icon path*. If you choose a specific icon, you have the following options via the buttons on the right side:



- **Insert and close:** The *virtual icon path* of the selected icon is inserted at the current cursor position in the script and the icon browser is closed.
- **Copy to clipboard:** The path is copied to the clipboard. The browser stays open.
- **Cancel:** Hides the dialog without further action.

5.3 Property Dialog

The property dialog stores the main itom settings, including all the widgets settings. The Property dialog can be found by clicking “File -> “Properties...”. It has different sections with subsections corresponding to the sections on this help page.

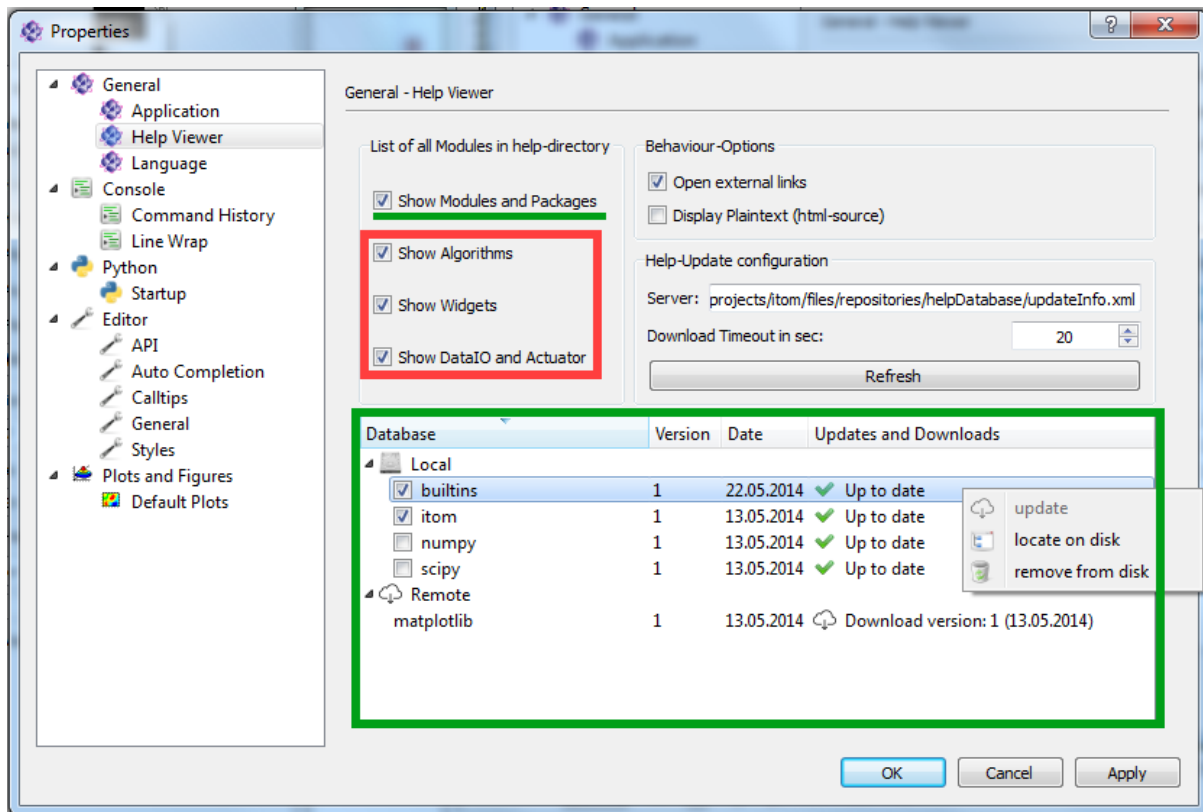
5.3.1 General

Application

- If the “show message before closing the application” checkbox is checked, the application will ask you if you really want to close Itom.

Help Viewer

This property section is responsible for the behaviour of the “Help” dialog. If the help widget is hidden in you mainwindow, go to View -> Toolboxes -> Help in the main toolbar.



Local and remote databases

Most help files are organized in databases. To display these files, the green underlined checkbox has to be checked. To manage, update and load new databases the green box offers a variety of options. Each database listed underneath “Local” are saved on the harddrive. The last column shows if there are any online updates available. To refresh the updatestate of the databases, just click the “refresh” button above.

If the the internet connection is very slow a timeout error might appear during updates. In this case increase the timeout time and check you internet connection.

Generated help files

The Algorithms, Widgets, DataIO and Actuator help files are dynamically created during runtime. These help files are displayed when the corresponding checkboxes in the red box are checked.

Language

By selecting a language inside the list box and clicking the “Apply” button a new language for itom is selected. Itom must be restarted to load the new language.

5.3.2 Console

Command History

These options refer to the command history widget that is available under View -> Toolboxes -> Command History.

Line Wrap

The first three radio buttons manage when a line is wrapped. The group box underneath offers the possibility to display small icons at a line wrap and to indent the next line. The lowest group box offers three modes how to indent the wrapped line.

5.3.3 Python

Startup

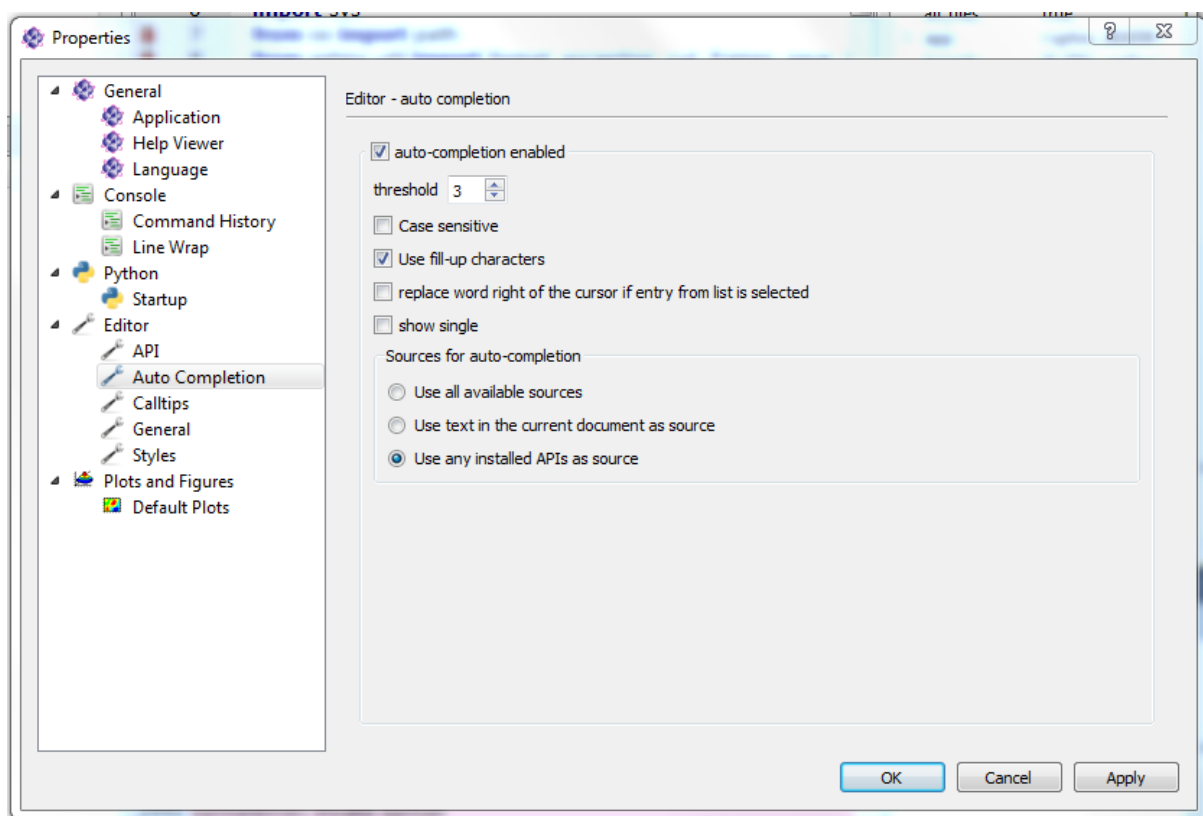
By “Add File” is possible to add python files that are executed when itom is started. It is kind of like the autostart folder in Windows.

5.3.4 Editor

API

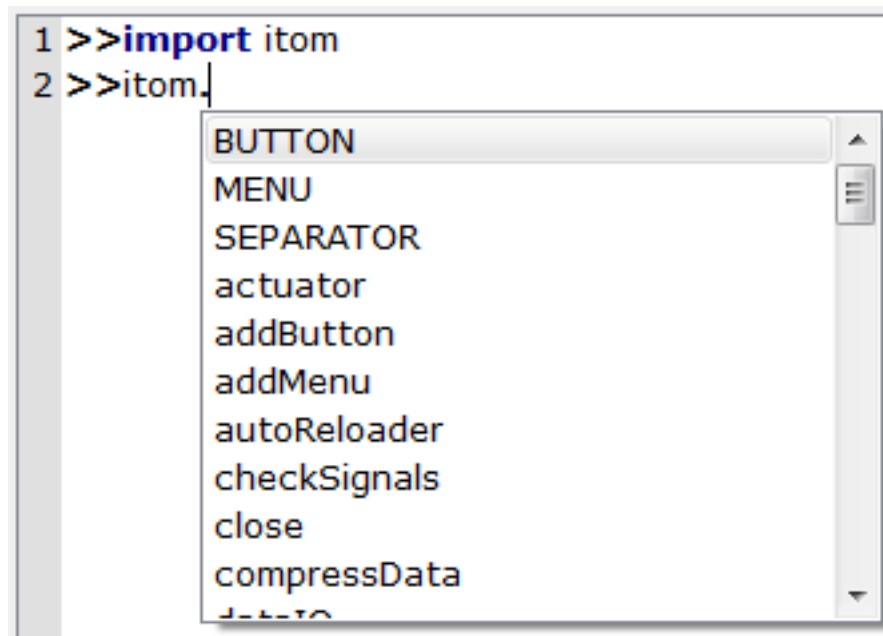
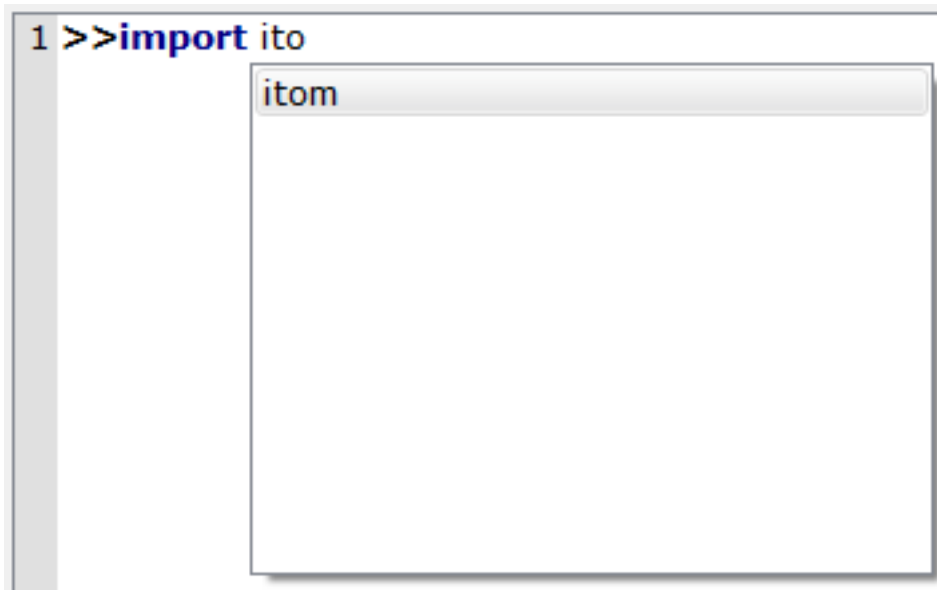
The api files listed in the checkbox are necessary for syntax highlighting. New api files can be added by clicking on the “Add API” button on the right side.

Auto Completion



The auto completion has two main functions. It offers available commands after entering some characters (number of minimum characters can be set in the “threshold” spin box).

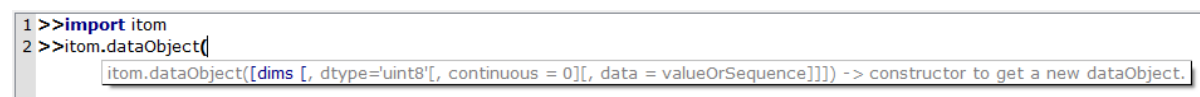
The other function shows a list of available members of classes after entering a dot.



The three radio buttons in the group box at the bottom of the page set the source of the auto completion. Therefore take a look at API.

Calltips

Calltips are tooltips that appear to display arguments of functions. They appear after entering “(”. The number of calltips is important if there are overloaded functions with different parameter sets.

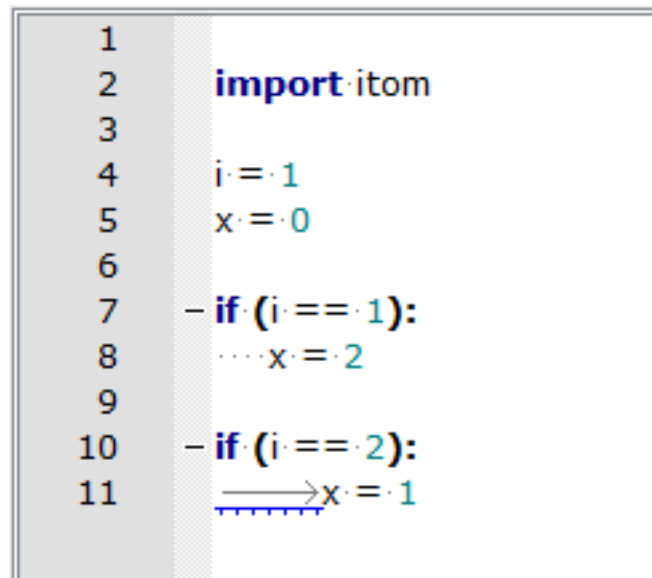


General

The first group box manages the indentation.

- “Auto indentation” automatically indents a new block after an “if ():” or a for loop occurred.
- if “Use tabs for indentation” is checked, tabs are used, otherwise spaces.
-
- “Show Whitespaces” displays small light grey dots in each indentation.
- The “Indentation Width” spinbox sets the standard width for the indentation

Inside “Indentation Warning” group box it is possible to select which kind of indentation is marked as wrong. Make sure not to create a conflict with the checkboxes listed above (“use Tabs for Indentation”). The following image shows a warning caused by wrong indentation (tabs).



The radio buttons inside the “End-of-line (EOL) mode” group box decide whether to use “\n”, “\r” or “\r\n” as eol, depending on your operating system.

The **Python Syntax Checker** checks the code inside the editor widget for bugs. If there are bug, a small red ladybug is shown besides the line numbers. If the cursor is moved over a ladybug, a tooltip shows the error (for more information see the help about the [script editor window](#)).

- The Itom module is always included in every script. This causes wrong bugs appearances because the checking module (frosted) is not able to see the itom inclusion. To avoid these errors check the “Automatically include itom module...” check box. It includes “include itom” in every header before checking the code to avoid wrong bugs.
- The “Check interval” check box sets the interval the code is send to “frosted” for syntax checks.

The **class navigator** feature allows configuring the [class navigator](#) of any script editor window. The checkbox of the entire groupbox en- or disables this feature. Use the timer to recheck the script structure after a certain amount of seconds since the last change of the script. If the timer is disabled, the structure is only analyzed when the script is shown or loaded.

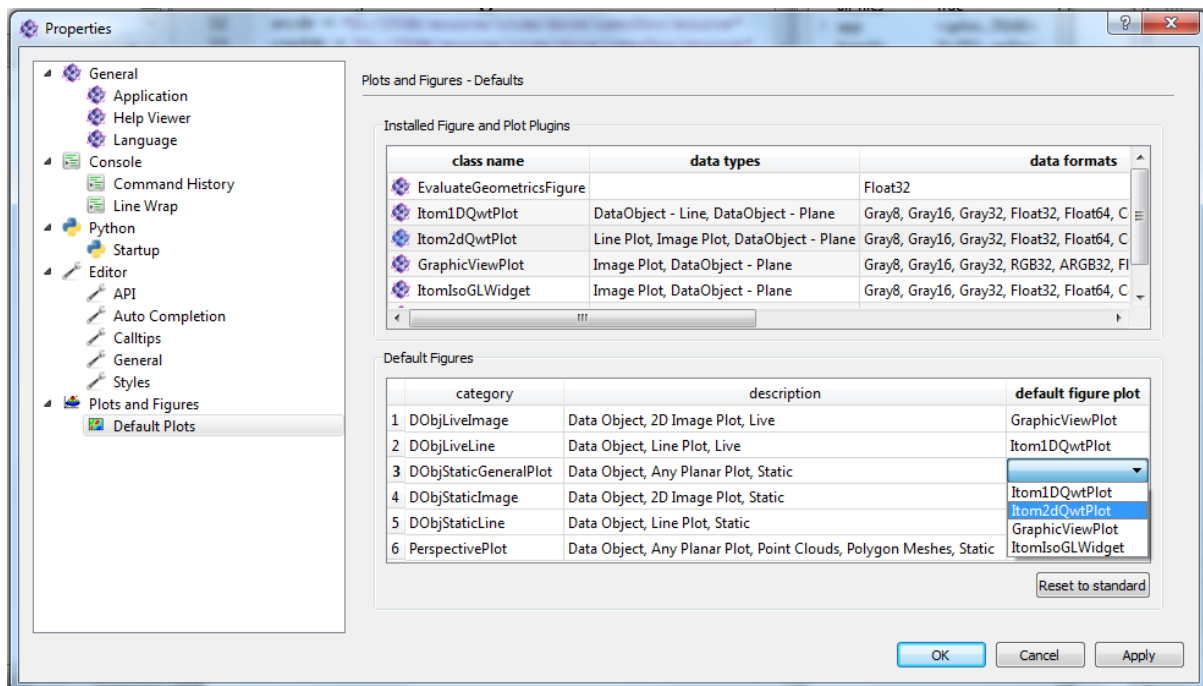
Styles

This page is responsible for the highlighting of reserved words, comments, identifier and so on. The style for each type of text, listed in the listbox, can be set individually.

5.3.5 Plots and Figures

Default Plots

- The first table lists all available plugins to plot data. The different columns show what kind of input data they accept and what they should be used for.
- The second table shows different categories for plots. For each category a default plugin can be selected. This default plugin will be used to plot the incoming data. To change the standard plugin, double click the last column.



5.4 Python Package Manager

Python is a very modular script language. Its functionality can be enhanced by so called packages. One package has a certain name and a version number, such that updates are possible. Furthermore, a package might dependent on multiple other packages that have to be installed in order to run the package.

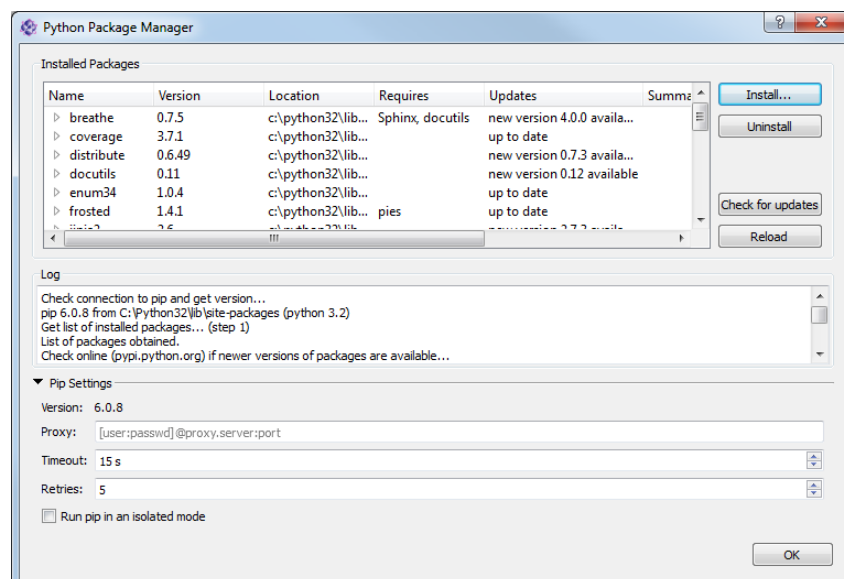
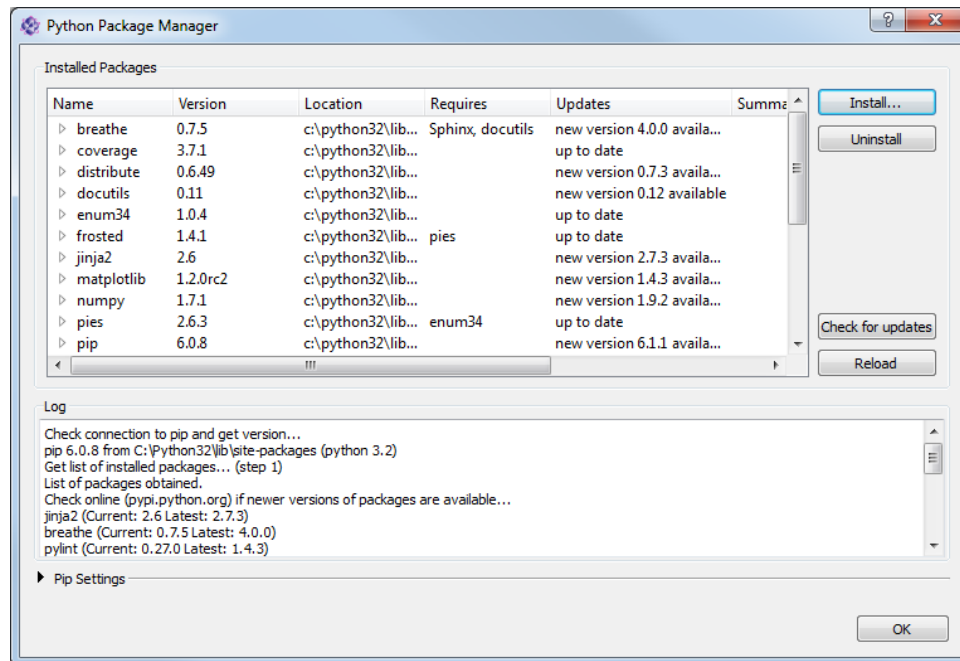
itom provides a package manager, that is accessible via the **Script** menu of the main window:

This package manager is a graphical user interface for the python package installation and administration tool **pip** (see <https://pip.pypa.io/>). Pip itself is a package of python and comes with a python installation from python 3.4 on. For earlier versions of python it is necessary to install pip first (see <https://pip.pypa.io/en/stable/installing.html#install-pip>).

Once the package manager is started, pip is detected and the version is checked. Then pip is asked to provide a list of currently installed packages that are listed in the list widget of the package manager. For this purpose an active internet connection is required. The connection can be further configured by enlarging the *pip settings* group box. All these configurations are also available when directly calling *pip* from the command line. The log window shows the output of pip. This output is finally parsed and leads to the content of the table *installed packages*. In case of a problem, text might also appear with red color in the log.

Depending on the version of pip, each package is described by

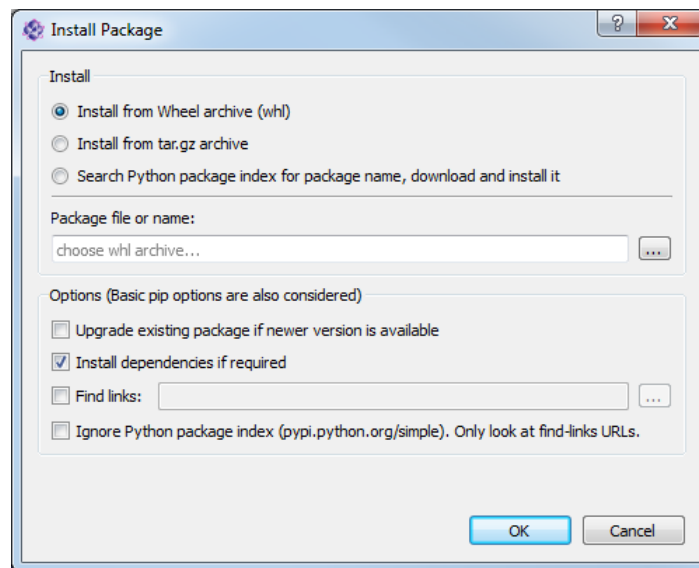
- its name
- its current version



- the location where the package is locally installed
- dependent packages that are required to run this package (comma-separated)
- information about possible updates in the internet (click *check for updates* to obtain this information)
- the package's summary
- a homepage about the package
- the package's license

Click the *check for updates* button, to screen the official python package index (<https://pypi.python.org>) for newer versions of the installed packages. *pip* itself is also listed as package.

If you want to install new packages or update an existing package, click the *install...* button:



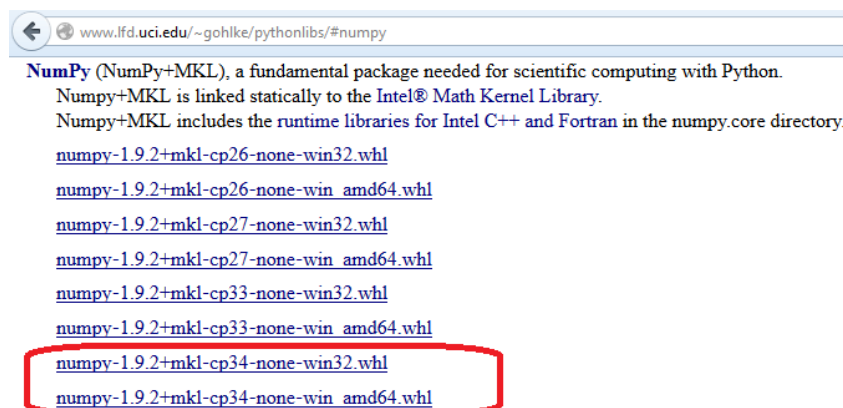
You have three possibilities to install a package. Choose if you want to install a package from a modern python wheel archive file (*whl*) or from a *tar.gz* archive. In both cases, chose the archive by clicking the tool button below. If you want to search the official python package index (<https://pypi.python.org>) for a specific package name and if found download and install, type the package name in the text field. See the next section for more information about the ideal package source.

If you want to upgrade an existing package, it is necessary to click the corresponding check box. If you want to also download all required packages, click the appropriate check box. Finally you can also indicate an alternative search location for packages using the *find links* option. Click OK to start the search and installation. Its progress is then shown in the log and the list of packages is updated at the end.

It is also possible to uninstall an installed package (however if no other package depends on this package). If desired, click a package and click then the *uninstall* button.

5.4.1 Package Sources

If you look for small packages (e.g. *frosted*), the easiest way of installing them is the direct download from the python package index. However, there exist also more complex packages that contain for instance a huge number of C files. These packages require a compiler on your computer, since the source code of the C files is compiled to binary files (that can be used by the python package) at installation time at your computer. Sometimes these packages also required further 3rd party libraries (e.g. Numpy, Scipy or Matplotlib). If so, it might be easier to directly get pre-compiled packages from the internet. One source for such precompiled packages is the website <http://www.lfd.uci.edu/~gohlke/pythonlibs/>:



The packages listed by this website are only available for Windows but different versions of Python in 32bit and 64bit. Download the corresponding wheel file and use the **itom** python package manager to install it (don't forget

to mark the upgrade check box if you want to update an existing package). An advantage of this site is also, that some libraries are compiled using the fast MKL compiler from Intel. Please notice that you also need to install updates for packages obtained from Gohlke by downloading the updated version as wheel file and install it. Do not mix the source from Gohlke and the python package index.

Under Linux, it is also worth and possible to install common packages, like Numpy, Scipy or Matplotlib from the package managers of your Linux distribution. This is the way to obtain prebuild binaries under Linux. The above mentioned website only provides Windows binaries.

PLOTS AND FIGURES

itom contains functionalities to plot the mainly supported data structures like data objects, compatible numpy arrays, point clouds or polygon meshes. Plots can be opened as an individual main window, they can be nested as subplots into one common figure or it is also possible to integrate plots into user-defined widgets and windows (see *Creating advanced dialogs and windows*).

In order to allow an integration of plots into user interfaces that are created by the external Qt Designer at runtime, every plot is appended by means of a plugin that implements the interface given by Qt Designer plugins. Therefore you can handle every itom plot as any other widget in Qt Designer, if this tool is called by itom.

Usually itom comes with a set of default plots, that can be used for the following plotting tasks:

- Static plot of a 1D data object (line plot).
- Static plot of a 2D or higher dimensional data objects, where the matrix values can be colorized by different color maps (default: grey map). For data objects with higher dimensions it is usually possible to change the currently depicted plane.
- Live plot of a line camera device (line plot).
- Live viewer for a camera providing two dimensional data objects (image plot).
- Three-dimensional plotting of 2D data objects where the data is interpreted as 2.5D, such that each value is considered to be a height value (isometric plot).
- There exist further plot plugins for visualizing an arbitrary number of point clouds, polygonal meshes or other geometrical features like spheres, cylinders or boxes.

See the following files in order to learn more about plots and figures in itom:

Content:

6.1 Quick tutorial to plots and figures

6.1.1 Plots of data objects

The most common way to plot an arbitrary data object is the `plot()` command contained in the module `itom`.

In the first example, we create an one-dimensional data object with random values (16bit, signed, fixed point precision) and then want to visualize this data object in a line plot. itom is able to recognize the type of plot you desire and uses the plot plugin which is set to be default for this type of plot (static, line plot). The defaults can be set in the *property dialog* of itom.

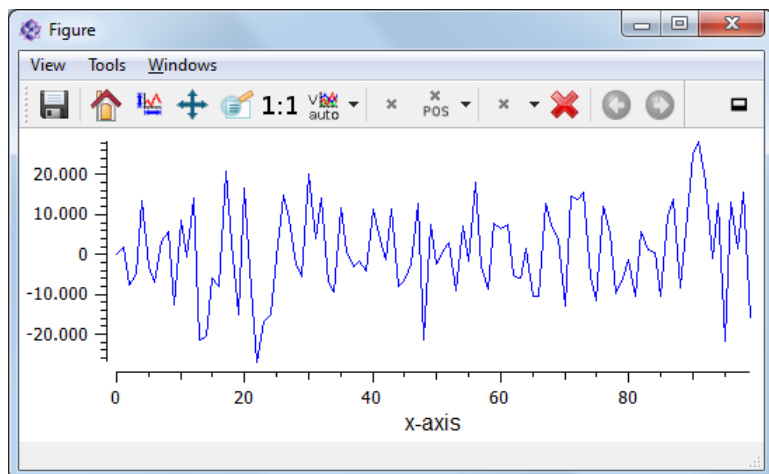
```
data1d = dataObject.randn([1,100], 'int16')
plot(data1d)
```

Note: Please consider that any one-dimensional data object is always exposed as two-dimensional data object, where the first (y) dimension is set to 1.

If you have various plot plugins available that can handle that type of data object, you can also force the plot command to use your specific plugin, which is defined by its class name (see itom's [property dialog](#) for the class name). If the class name cannot be found or if it is not able to plot the type of data object, itom falls back to the default plot plugin (and prints a warning into the console):

```
plot(data1d, "itom1DQwtPlot") #case insensitive plot class name
```

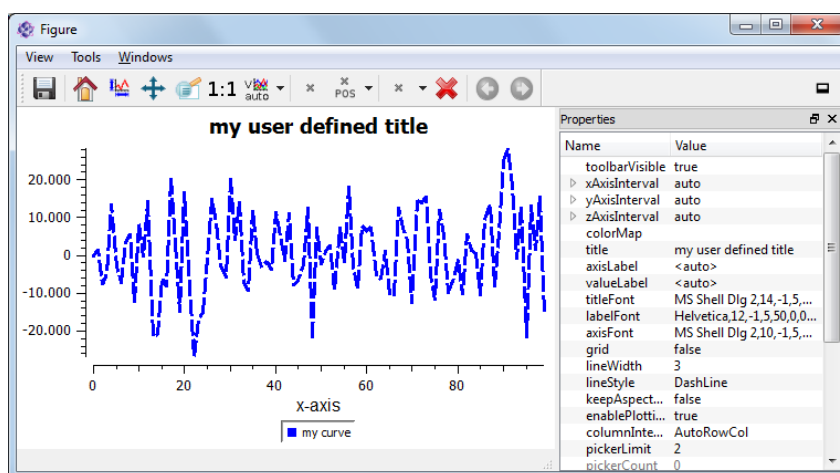
The result of both examples looks like this (if no other default plot class has been chosen for 1D static plots):



In the following sections, you will see that any plot has various properties that can be set in the property dialog or using square brackets in Python. However, you can also pass various properties to the `plot()` command such that your customized plot is displayed.

```
plot(data1d, properties={"title":"my user defined title","lineWidth":3, \
    "lineStyle":"DashLine","legendPosition":"Bottom", \
    "legendTitles":"my curve"})
```

Then, the plot looks like thies:

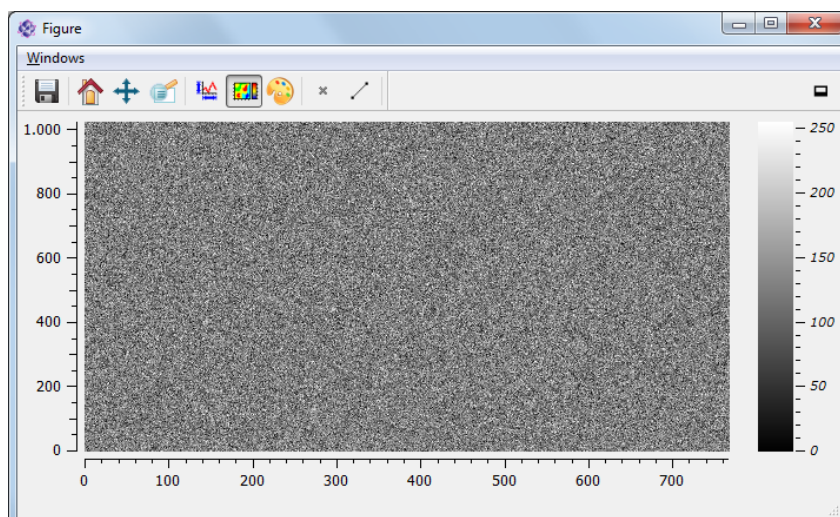


Passing a dictionary with various properties works with all types of plots. However, the list of available properties might change and can be obtained either using the Python command `info()` or displaying the properties toolbox of the plot. For more information see also [Properties of plots](#) below.

Equivalent to the one-dimensional case, the following example shows how to simply plot a two-dimensional data object also using the command `plot()`.

```
data2d = dataObject.randn([1024,768],'uint8')
plot(data2d)
```


Then, you obtain a figure that looks like this:



If you not only work with data objects but also with numpy you can also pass numpy arrays to the `plot()` command. An implicit shallow copy in terms of a `itom.dataObject` is then created and passed to the plots.

If the plot is opened in its own figure window, you have a dock-button in the toolbar on the right side. Click on this button in order to dock the plot into the main window of itom.

6.1.2 Live images of cameras and grabbers

itom is not only able to plot data objects but can also show live streams of connected and opened cameras. Cameras are implemented as plugins of type `dataIO` that also have the `grabber-type` flag defined (see the section grabbers of your *plugin toolbox* in itom). If a live image of a specific camera should be created, the following process is started:

1. The camera is asked for its parameters `sizeX` and `sizeY`. If one of these dimensions is equal to one, a live line image is opened, else a two-dimensional live image is opened.
2. The command `startDevice()` of the camera is called (idle command if the camera is already started)
3. A timer continuously triggers the image acquisition of the camera and sends the result to all currently connected live images. However the timer is not started or stopped whenever the auto-grabbing property of the camera is disabled. This is useful, if you are in the middle of measurement process. Then you don't want the timer to force the image acquisition but your process. Therefore, disable to auto-grabbing property before starting your measurement and reset it to its previous status afterwards. In any case, whenever any process triggers an image acquisition, all results will always be sent to connected live images.
4. When the live plot is closed or disconnected, the command `itom.dataIO.stopDevice()` is called (this is again an idle command if the camera is still used by other live images or has been started by any python script and not stopped yet).

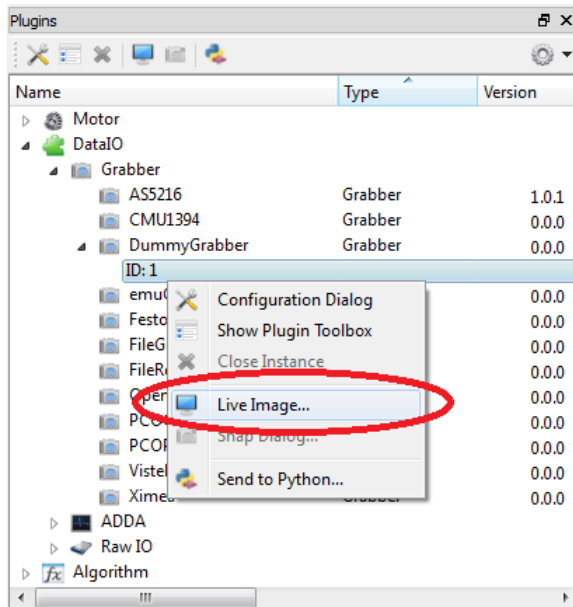
In the following example, the dummy grabber camera is started and the live image is opened using the command `liveImage()`. The auto-grabbing property is set to `True` (which is also the default case):

```
cam = dataIO("DummyGrabber")
cam.setAutoGrabbing(True) #can be omitted if auto grabbing already enabled
liveImage(cam)
```

You can also show the live image of any camera using the GUI. Right-click on the opened camera instance in the plugin toolbox and choose **live image**:

6.1.3 Properties of plots

Any plots have properties defined, which indicate the appearance or currently depicted data object or camera. To access these cameras you need to get the instance of the plot or live image item. This is always an instance of the



class `plotItem`. This class is inherited by `uiItem` which finally provides the access to the properties by the functionalities described in [Creating advanced dialogs and windows](#).

In order to access the necessary instance of `plotItem`, you will see that the return value of the commands `plot()` or `liveImage()` is a tuple consisting of a number of the overall figure (window), where the plot is print and of the requested instance as second value.

In the next example, the title of a two-dimensional data object plot is changed:

```
data2d = dataObject.randN([100,100])
[idx,h] = plot(data2d)
h["title"] = "new title"
```

Note: Not all plot plugins have the same properties defined, since this also depends on their type and special features. However it is intended to use the same property names for the same meaning in the different plugins.

Note: If the figure closed while you still have a reference to its instance, any method of this instance will raise an error saying that the plot does not exist any more.

In order to get a list of all properties of a plot, call the method `info()` of the plot instance. This method prints a list of available properties as well as slots and signals.

```
h.info()
```

There are two other important properties that let you change the displayed data object or camera:

```
#set new data object
h["source"] = dataObject.randN([100,100])

#assign new camera
h["camera"] = dataIO("DummyGrabber")
```

These properties are also the way to set the content of plot widgets, that are integrated in your user-defined GUIs.

The properties can also be changed using the properties toolbox of each plot or live image that is accessible via the menu `View >> Properties`. Furthermore it is possible to directly set some properties by passing a dictionary with all name, values pairs to the 'properties' argument of commands `plot()` or `liveImage()`:

```
plot(data2d, properties={"yAxisFlipped":True, "title":"My self configured plot"})
```

6.2 Figure Management

Technically, every plot widget, hence the plot containing the axes and the content, is represented as an instance of `plotItem`. If you show the plot in its own window, this window is called figure and is represented as an instance of `figure`. Every figure is also able to show various sub-plots arranged in a regular grid of m rows and n columns. Plots, integrated in an user-defined interface, are directly integrated there without being part of a special figure.

If you want to continuously use the same figure window for sequentially plotting different data or if you want to use the sub-plot mechanism, it is useful to firstly create or access a specific figure and then plot data objects or camera live images using the specific commands of this figure.

Every figure has a specific handle (integer value). Therefore you can have access to any figure by its figure instance or via the handle value.

In the following example, a new figure is created and opened. The instance is saved in the variable `fig`:

```
fig = figure()
```

Assuming, that you want to access an already existing figure or create a new figure with your desired handle number, give the handle as first argument (keyword: `handle`):

```
fig = figure(1)
#or
fig = figure(handle = 1)
```

`fig` is then an instance of `figure`. This class is derived from `uiItem` and therefore has the same functionality like any other widget that is represented by `uiItem` (see [Creating advanced dialogs and windows](#)). If you provide no further parameter (besides `handle`) to the figure constructor, a figure with one plot area is created. If you want to create a grid of more plot areas, you need to give the number of rows and columns as argument to the constructor:

```
fig = figure(rows = 2, cols = 3)
```

Not only the module `itom` has the methods `liveImage()` in order to plot data objects or camera streams. In the same way, the class `figure` has these methods, too:

With method signature:

<code>itom.figure.plot((data, [areaIndex, ...)</code>	Plot an existing dataObject in not dockable, not blocking window.
<code>itom.figure.liveImage((cam, [areaIndex, ...)</code>	Creates a plot-image (2D) and automatically grabs images into this window.

6.2.1 itom.figure.plot

`figure.plot(data[, areaIndex, className, properties])` → plots a dataObject in the current or given area of this figure

Plot an existing dataObject in not dockable, not blocking window. The style of the plot will depend on the object dimensions. If x-dim or y-dim are equal to 1, plot will be a lineplot else a 2D-plot.

Parameters `data` : {DataObject}

Is the data object whose region of interest will be plotted.

areaIndex: {int}, optional :

Area number where the plot should be put if subplots have been created

className : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings)

properties : {dict}, optional

optional dictionary of properties that will be directly applied to the plot widget.

6.2.2 itom.figure.liveImage

`figure.liveImage(cam[, areaIndex, className, properties])` → shows a camera live image in the current or given area of this figure
Creates a plot-image (2D) and automatically grabs images into this window. This function is not blocking.

Parameters `cam` : {dataIO-Instance}

Camera grabber device from which images are acquired.

areaIndex: {int}, optional :

Area number where the plot should be put if subplots have been created

className : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

properties : {dict}, optional

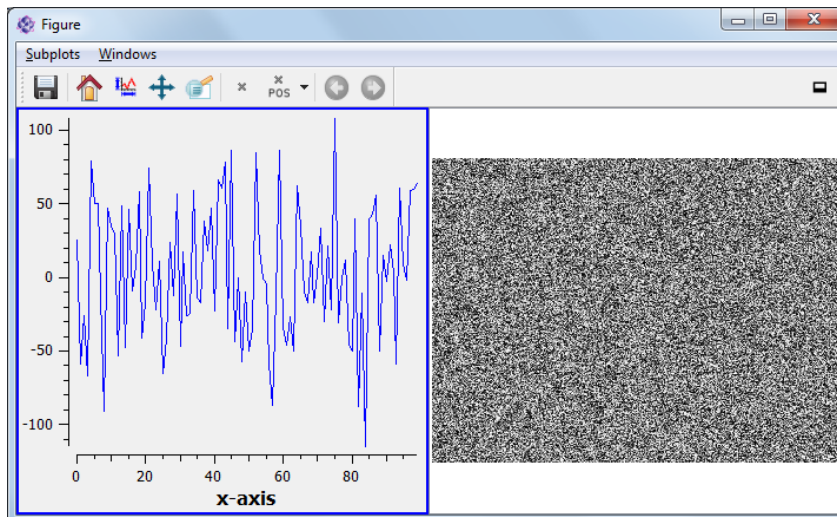
optional dictionary of properties that will be directly applied to the plot widget.

Both have the optional keyword parameter *areaIndex*, such that you define in which sub-plotting area the requested plot or live image should be depicted. If you don't provide the *areaIndex* parameter, the first area (top, left) is assumed (this is also the default in case that you don't use subplots). The *areaIndex* value is a fixed-point number beginning with 0 for the top-left area of the figure. Then, the index iterates row-wise through the entire grid of sub-plots.

In the following example, a 1x2 grid is created where the left plot shows a 1D data object and the right plot contains the live stream of the dummy grabber camera instance:

```
fig = figure(rows = 1, cols = 2)
data1d = dataObject.randn([1,100])
fig.plot(data1d, 0)
cam = dataIO("DummyGrabber")
fig.liveImage(cam, 1)
```

The result looks like this:



The currently depicted toolbar is provided by the active subplot, marked by a border. You can switch the active subplot by choosing your desired one in the figure's menu *subplot*. In order to get the instance of *itom.plotItem* of any subplot, use the following command:

```
plotLeft = fig.subplot(0)
plotRight = fig.subplot(1)
```

Note: If you have an instance to a plotItem only in python and the window is closed, the window is finally deleted and the plotItem becomes invalid. In difference, a handle to a figure keeps the figure alive until all references to the figure's instance are deleted. Therefore you can always show or hide a figure using its instance and the methods `show()` or `hide()`.

The `figure` also provides a static method `close()` to close and delete a figure defined by its specific integer handle or to close all figures using the string 'all' as parameter. Please note, that only these figures are finally deleted where no other python references exist to them.

```
#close figure with handle 7
figure.close(7)

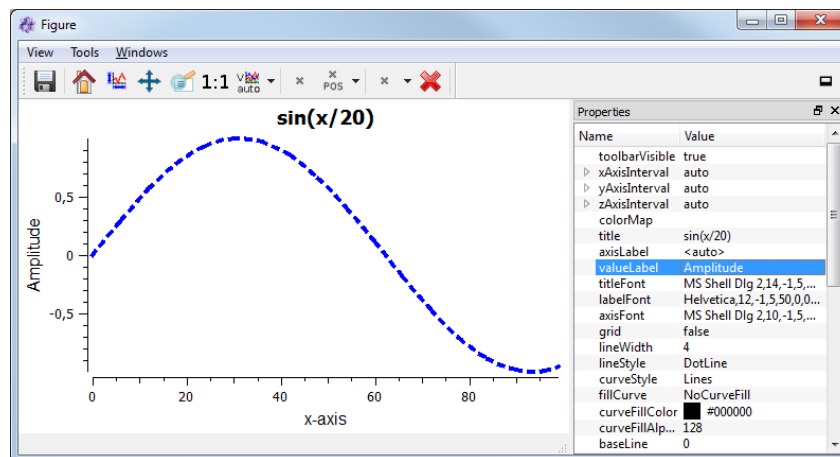
#close all figures
figure.close('all')
```

6.3 line plots (1D)

In order to plot a 1D line (1 x M or M x 1 DataObject) or multiple lines in a line plot, the designer plugin **Itom1DQwtPlot** is the recommended visualization tool. You can either add this class name (`itom1dqwtplot`) to any `plot()` or `liveImage()` command in order to force the data to be plotted in this plot type and / or set this plugin to be the default plot for 1D static and live plots.

The latter can be done in the property dialog or itom, tab **Plots and Figures >> Default Plots**. Set there the *default figure plot* to `Itom1DQwtPlot` for the categories

- DObjLiveLine
- DObjStaticLine



This plot has been created by the following code snippet:

```
import numpy as np
X = np.linspace(0,100)
Y = np.sin(X/20)
plot(Y, className = 'itom1dqwtplot', \
     properties = {"title":"sin(X/20)", "valueLabel": "Amplitude"})
```

If you choose `itom1dqwtplot` as `className` for the `plot()` command with a 2D data object as argument, is it also possible to plot multiple lines. The plot plugin accepts all available data types, including colors and complex values.

Data is plotted as follows:

- Real data types: One or multiple lines are plotted where the horizontal axis corresponds to the grid of the data object considering possible scaling and offset values. The line(s) have different colours. The colour of one line can also be adjusted.
- Complex types: This is the same than for real data types, however you can choose whether the *absolute*, *phase*, *real* or *imaginary* part of the complex values is plotted.
- color type: Coloured data objects will be represented by either 3 or 4 lines (red, green and blue, alpha optional) that correspond to the three colour channels or by one line representing the converted gray values.

Using Python or the properties toolbox (View >> properties or via right click on the toolbar), it is possible to adjust many properties like stated below.

The plot allows value and min/max-picking via place-able marker.

The plot supports geometric element and marker interaction via **drawAndPickElements(...)** and **call("userInteractionStart",...)**. See section [Primitives - Marking and Measuring](#) for a short introduction.

You can also use the “matplotlib”-backend to plot slices or xy-coordinates. See section [Python-Module matplotlib](#) for more information about how to use “matplotlib”.

The plot-canvas can be exported to vector and bitmap-graphics via button or menu entry or it can be exported to clipBoard via ctrl-c or a public slot.

6.3.1 Properties

selectedGeometry : *int*, get the currently selected geometric element within this plot

enablePlotting : *bool*, enable and disable internal plotting functions and GUI-elements for geometric elements.

keepAspectRatio : *bool*, enable and disable a fixed 1:1 aspect ratio between x and y axis.

geometricElementsCount : *int*, number of currently existing geometric elements.

geometricElements : *ito::DataObject*, geometric elements defined by a float32[11] array for each element.

axisFont : *QFont*, font for axes tick values.

labelFont : *QFont*, Font for axes descriptions.

titleFont : *QFont*, Font for title.

valueLabel : *QString*, Label of the value axis (y-axis) or ‘<auto>’ if the description should be used from data object.

axisLabel: *QString*, Label of the direction (x/y) axis or ‘<auto>’ if the descriptions from the data object should be used.

title : *QString*, Title of the plot or ‘<auto>’ if the title of the data object should be used.

bounds : *QVector<QPointF>*,

colorMap : *QString*, Color map (string) that should be used to colorize a non-color data object.

yAxisInterval : If member *auto* of *autoInterval* is False, the visible range of the displayed y-axis is set to the given range (in coordinates of the data object); else the range is automatically determined and set [default].

xAxisInterval : If member *auto* of *autoInterval* is False, the visible range of the displayed x-axis is set to the given range (in coordinates of the data object); else the range is automatically determined and set [default].

camera : *ito::AddInDataIO*, Use this property to set a camera/grabber to this plot (live image).

displayed : *ito::DataObject*, This returns the currently displayed data object [read only].

source : *ito::DataObject*, Sets the input data object for this plot.

contextMenuEnabled : *bool*, Defines whether the context menu of the plot should be enabled or not.

toolbarVisible : *bool*, Toggles the visibility of the toolbar of the plot.

6.3.2 Signals

plotItemsFinished(int,bool): Signal emitted if plotting of n-elements if finished. Use this for non-blocking synchronisation.

counts, int: Number of plotted elements

aborted, bool: Flag showing if draw function was cancelled during plotting

plotItemsDeleted():

Signal emitted if geometric elements were deleted.

plotItemDeleted(ito::int32):

Signal emitted if specified geometric element was deleted.

plotItemChanged(ito::int32,ito::int32,QVector<ito::float32>):

Signal emitted if specified geometric element was changed.

idx, ito::int32: Index of changed element

element, QVector<ito::float32>: New geometric featured of changed element

userInteractionDone(int,bool,QPolygonF):

Signal emitted if user interaction is done. Internal function used for blocking synchronisation.

6.3.3 Slots

ito::DataObject getDisplayed():

Retrieve currently displayed dataObject.

ito::RetVal clearGeometricElements():

Delete all geometric Elements

void userInteractionStart(int type, bool start [, int maxNrOfPoints = -1]):

This slot should be called of non-blocking GUI-based drawing of geometric elements within this plot is necessary. See section *Primitives - Marking and Measuring* for a short introduction.

type, int: type to plot

start, bool: true if plotting should be started

maxNrOfPoints, int: number of elements to plot

ito::RetVal deleteMarkers(int id):

Delete geometric element

id, int: the 0-based index of specific geometric element

ito::RetVal plotMarkers(ito::DataObject coords, QString style [, QString id = "" [, int plane = -1]]):

This slot is called to visualize markers and python-based plotting of geometric elements within this plot. See section *Primitives - Marking and Measuring* for a short introduction.

coords, ito::DataObject: an initialized dataObject with a column per element and a set of rows describing its geometric features

style, QString: Style for plotted markers, for geometric elements it is ignored

id, QString: Text based id for markers will be ignored for geometric elements.

ito::RetVal setLinePlot(double x0, double y0, double x1, double y1 [, int linePlotIdx = -1]):

this can be invoked by python to trigger a line plot, inherited from *class AbstractDObjFigure*

x0, double: first position of line plot in x-Direction

y0, double: first position of line plot in y-Direction

x1, double: second position of line plot in x-Direction

y1, double: second position of line plot in x-Direction

ito::RetVal setSource(ito::DataObject source, ItomSharedSemaphore*)

Set new source object to this plot. Usually invoked by any camera if used as a live image.

**source, ito::DataObject **: The new dataObject to display

semaphore, ItomSharedSemaphore: A semaphore to handle the multi-threading.

refreshPlot():

Refresh / redraw current plot

copyToClipboard():

Copy current canvas with white background to clipBoard

Deprecated figures

The plot-dll “itom1DQWTFigure” is deprecated and has been replaced by “Itom1DQwtPlot”.

6.4 2D image plots

“Itom2dQwtPlot” and “GraphicViewPlot” are the basic plots for visualization of images, dataObjects or other array-like objects. Both plots have a line-cut and point picker included. By pressing “Ctrl” during picker movement, the picker can only be moved horizontal or vertical according to the mouse movement.

You can also use the “matplotlib”-backend to plot any data structures (lines, bars, statistical plots, images, contours, 3d plots...). See section *Python-Module matplotlib* for more information about how to use “matplotlib”.

6.4.1 Itom2dQwtPlot

“Itom2dQwtPlot” is designed for visualizing metrical data, false color or topography measurements. It supports the axis-scaling / axis offset of **dataObjects**, offers axis-tags and meta-data handling. All data types are accepted except the plotting of real color objects (rgba). To plot complex objects, it is possible to choose between the following modes: “absolute”, “phase”, “real” and “imaginary”. The data is plotted mathematically correct. This means the value at [0,0] is in the lower left position. This can be changed by the property *yAxisFlipped*.

The plot supports geometric element and marker interaction via **drawAndPickElements(...)** and **call(“userInteractionStart”,...)**. See section *Primitives - Marking and Measuring* for a short introduction.

Features:

- Export graphics to images, pdf, vector graphics (via button) or to the clipboard (ctrl-c).
- Metadata support (the ‘title’-tag is used as title of the plot).
- Supports fixed ratio x/y-axis but not necessary fixed ratio to monitor-pixel
- Drawing of geometrical elements and markers by script and user interaction.
- Images are displayed either mathematically ([0,0] lower left) or in windows-style ([0,0] upper left) (Property: ‘yAxisFlipped’)

Properties

selectedGeometry : *int*, Get or set the currently highlighted geometric element. After manipulation the last element stays selected.

showCenterMarker : *bool*, Enable a marker for the center of a data object.

enablePlotting : *bool*, Enable and disable internal plotting functions and GUI-elements for geometric elements.

keepAspectRatio : *bool*, Enable and disable a fixed 1:1 aspect ratio between x and y axis.

geometricElementsCount : **int*, Number of currently existing geometric elements.

geometricElements : *ito::DataObject*, Geometric elements defined by a float32[11] array for each element.

axisFont : *QFont*, Font for axes tick values.

labelFont : *QFont*, Font for axes descriptions.

titleFont : *QFont*, Font for title.

colorMap : *QString*, Defines which color map should be used [e.g. grayMarked, hotIron].

colorBarVisible : *bool*, Defines whether the color bar should be visible.

valueLabel : *QString*, Label of the value axis or '<auto>' if the description should be used from data object.

yAxisFlipped : *bool*, Sets whether y-axis should be flipped (default: false, zero is at the bottom).

yAxisVisible : *bool*, Sets visibility of the y-axis.

yAxisLabel : *QString*, Label of the y-axis or '<auto>' if the description from the data object should be used.

xAxisVisible : *bool*, Sets visibility of the x-axis.

xAxisLabel : *QString*, Label of the x-axis or '<auto>' if the description from the data object should be used.

title : *QString*, Title of the plot or '<auto>' if the title of the data object should be used.

zAxisInterval : *QPointF*, Sets the visible range of the displayed z-axis (in coordinates of the data object) or (0.0, 0.0) if range should be automatically set [default].

yAxisInterval : *QPointF*, Sets the visible range of the displayed y-axis (in coordinates of the data object) or (0.0, 0.0) if range should be automatically set [default].

xAxisInterval : *QPointF*, Sets the visible range of the displayed x-axis (in coordinates of the data object) or (0.0, 0.0) if range should be automatically set [default].

camera : *ito::AddInDataIO*, Use this property to set a camera/grabber to this plot (live image).

displayed : *ito::DataObject*, This returns the currently displayed data object [read only].

source : *ito::DataObject*, Sets the input data object for this plot.

contextMenuEnabled : *bool*, Defines whether the context menu of the plot should be enabled or not.

toolbarVisible : *bool*, Toggles the visibility of the toolbar of the plot.

Signals

plotItemsFinished(int,bool): Signal emitted if plotting of n-elements is finished. Use this for non-blocking synchronisation.

counts, *int*: Number of plotted elements

aborted, *bool*: Flag showing if draw function was cancelled during plotting

plotItemsDeleted():

Signal emitted if geometric elements were deleted.

plotItemDeleted(ito::int32):

Signal emitted if specified geometric element was deleted.

plotItemChanged(itom::int32, itom::int32, QVector<itom::float32>):

Signal emitted if specified geometric element was changed.

idx, itom::int32: Index of changed element

element, QVector<itom::float32>: New geometric featured of changed element

userInteractionDone(int, bool, QPolygonF):

Signal emitted if user interaction is done. Internal function used for blocking synchronisation.

Slots

itom::DataObject getDisplayed():

Retrieve currently displayed dataObject.

itom::RetVal clearGeometricElements():

Delete all geometric Elements

void userInteractionStart(int type, bool start [, int maxNrOfPoints = -1]):

This slot should be called of non-blocking GUI-based drawing of geometric elements within this plot is necessary. See section *Primitives - Marking and Measuring* for a short introduction.

type, int: type to plot

start, bool: true if plotting should be started

maxNrOfPoints, int: number of elements to plot

itom::RetVal deleteMarkers(int id):

Delete geometric element

id, int: the 0-based index of specific geometric element

itom::RetVal deleteMarkers(QString id):

Delete point based marker

id, QString: the name based identifier of specific geometric element

itom::RetVal plotMarkers(itom::DataObject coords, QString style [, QString id = "" [, int plane = -1]]):

This slot is called to visualize markers and python-based plotting of geometric elements within this plot. See section *Primitives - Marking and Measuring* for a short introduction.

coords, itom::DataObject: an initialized dataObject with a column per element and a set of rows describing its geometric features

style, QString: Style for plotted markers, for geometric elements it is ignored

id, QString: Text based id for markers will be ignored for geometric elements.

itom::RetVal setLinePlot(double x0, double y0, double x1, double y1 [, int linePlotIdx = -1]):

this can be invoked by python to trigger a line plot, inherited from *class AbstractDObjFigure*

x0, double: first position of linePlot in x-Direction

y0, double: first position of linePlot in y-Direction

x1, double: second position of linePlot in x-Direction

y1, double: second position of linePlot in x-Direction

itom::RetVal setSource(itom::DataObject source, ItomSharedSemaphore*)

Set new source object to this plot. Usually invoked by any camera if used as a live image from internal C++-Code.

***source, ito::DataObject ***: The new dataObject to display

semaphore, ItomSharedSemaphore: A semaphore to handle the multi-threading.

refreshPlot():

Refresh / redraw current plot

copyToClipboard():

Copy current canvas with white background to clipBoard

6.4.2 GraphicViewPlot

“GraphicViewPlot” is designed for the fast display of images, e.g. direct grabber output or colored images. It allows plotting real colors (at the moment only 24-bit or 32-bit stored as int32 or RGBA32). It does not handle meta-data. All DataTypes are accepted. To plot complex objects, it is possible to select between the following modes: “absolute”, “phase”, “real” and “imaginary”. The data is plotted image orientated. This means the value at [0,0] is in the upper left position.

The figure allows z-stack sectioning. An automatic video-like visualisation is in preparation.

The “GraphicViewPlot” does not support graphic element / marker plotting. Use “Itom2dQwtPlot” instead for this case.

Features:

- Supports real color and grey-value visualization
- Supports fixed ratio between image-pixel and monitor-pixel (4:1 - 1:4)
- Fast implementation for 8-bit and 16-bit direct camera output.
- Images are displayed in windows-style

Properties

colorMap : *QString*, Color map (string) that should be used to colorize a non-color data object [e.g. grayMarked, hotIron].

colorBarVisible : *bool*, Defines whether the color bar should be visible.

colorMode : *int*, Defines color handling, either “palette-based color” or “RGB-color”

zAxisInterval : *autoInterval*. If member *auto* of *autoInterval* is False, the visible range of the displayed z-axis is set to the given range (in coordinates of the data object); else the range is automatically determined and set [default].

yAxisInterval : If member *auto* of *autoInterval* is False, the visible range of the displayed y-axis is set to the given range (in coordinates of the data object); else the range is automatically determined and set [default].

xAxisInterval : If member *auto* of *autoInterval* is False, the visible range of the displayed x-axis is set to the given range (in coordinates of the data object); else the range is automatically determined and set [default].

camera : *ito::AddInDataIO*, Use this property to set a camera/grabber to this plot (live image).

displayed : *ito::DataObject*, This returns the currently displayed data object [read only].

source : *ito::DataObject*, Sets the input data object for this plot.

contextMenuEnabled : *bool*, Defines whether the context menu of the plot should be enabled or not.

toolbarVisible : *bool*, Toggles the visibility of the toolbar of the plot.

Slots

ito::RetVal setLinePlot(double x0, double y0, double x1, double y1 [, int linePlotIdx = -1]):

this can be invoked by python to trigger a line plot, inherited from *class AbstractDObjFigure*, not implemented at the moment

x0, double: first position of line plot in x-Direction

y0, double: first position of line plot in y-Direction

x1, double: second position of line plot in x-Direction

y1, double: second position of line plot in x-Direction

ito::RetVal setSource(ito::DataObject source, ItomSharedSemaphore*)

Set new source object to this plot. Usually invoked by any camera if used as a live image from internal C++-Code.

**source, ito::DataObject **: The new dataObject to display

semaphore, ItomSharedSemaphore: A semaphore to handle the multi-threading.

refreshPlot():

Refresh / redraw current plot

Signals

No public signals at the moment.

6.4.3 Deprecated figures

The plot-dll “itom2DQWTFigure” and “itom2DGVFigure” are deprecated and have been replaced by “Itom2dQwtPlot” and “GraphicViewPlot”.

6.5 isometric Plot

“ItomIsoGLWidget” is a plot for pseudo 3D visualization of image like DataObjects. It is based on openGL and renders the objects either to triangles (“triangle mode”) or points (“Joe-Mode”). All DataTypes except “rgba32” are accepted. To plot complex objects, it is possible to select between the following modes: “absolute”, “phase”, “real” and “imaginary”.

The figure does not support z-stack sectioning. The “ItomIsoGLWidget” does support neither graphic element / marker plotting nor line or pixel picking. Hence this plot will be improved and replaced by a new version for the next release.

6.5.1 Properties

colorMap : *QString*, Defines which color map should be used [e.g. grayMarked, hotIron].

zAxisInterval : *QPointF*, Sets the visible range of the displayed z-axis (in coordinates of the data object) or (0.0, 0.0) if range should be automatically set [default]. **Not implemented yet**

yAxisInterval : *QPointF*, Sets the visible range of the displayed y-axis (in coordinates of the data object) or (0.0, 0.0) if range should be automatically set [default]. **Not implemented yet**

xAxisInterval : *QPointF*, Sets the visible range of the displayed x-axis (in coordinates of the data object) or (0.0, 0.0) if range should be automatically set [default]. **Not implemented yet**

camera : *ito::AddInDataIO*, Use this property to set a camera/grabber to this plot (live image).

displayed : *ito::DataObject*, This returns the currently displayed data object [read only].

source : *ito::DataObject*, Sets the input data object for this plot.

contextMenuEnabled : *bool*, Defines whether the context menu of the plot should be enabled or not. **Not implemented yet**

toolbarVisible : *bool*, Toggles the visibility of the toolbar of the plot. **Not implemented yet**

6.5.2 Slots

ito::RetVal setLinePlot(double x0, double y0, double x1, double y1 [, int linePlotIdx = -1]):

this can be invoked by python to trigger a line plot, inherited from *class AbstractDObjFigure*, **not implemented at the moment**

x0, double: first position of line plot in x-Direction

y0, double: first position of line plot in y-Direction

x1, double: second position of line plot in x-Direction

y1, double: second position of line plot in x-Direction

ito::RetVal setSource(ito::DataObject source, ItomSharedSemaphore*)

Set new source object to this plot. Usually invoked by any camera if used as a live image from **internal C++-Code**.

***source, ito::DataObject ***: The new dataObject to display

semaphore, ItomSharedSemaphore: A semaphore to handle the multi-threading.

refreshPlot():

Refresh / redraw current plot

triggerReplot():

Refresh / redraw current plot

Deprecated figures

None

6.6 Primitives - Marking and Measuring

The plot-widgets *itom1DQwtPlot* and *itom2DQwtPlot* supports plotting of geometric primitives by user interaction and script language. This section will give a short introduction about plotting, read- /write-functions and the corresponding plots and the internal geometric element structure.

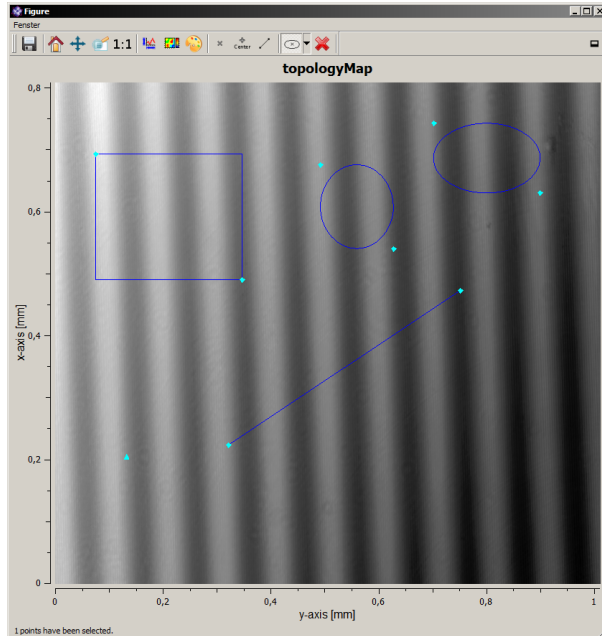
At last the *evaluateGeomtrics*-plugin for direct evaluation of geometric elements is introduced.

6.6.1 Drawing items into a Qwt-Plot

The plot functionality can be accessed by three different ways. The first way is the GUI based approach by which the user presses the “switch draw mode”-button in the button-bar of the plot. The button represents the current item to be plotted. The red X (“clear button”) will delete all geometric elements within the plot either drawn by hand or by script.



At the moment “itom” only supports “point”, “line”, “rectangle” and “ellipse” but further items, e.g. “circle” and “polygons”, are in preparation. To draw an item simply click into the image space and left-click the mouse. In case of elements with at least more than a marker, you can now set the size of the element by setting the second point by left-clicking again. During plotting a green lined geometric element appears. After finishing the element color turns to the first inverse color of the current color palette with handles (diamonds or squares) colored with the second inverse color of the current palette.



After creation the geometric elements can be edited by left-clicking one of the element handles which becomes high-lighted (squares) and moving the mouse. By pressing the “ctrl”-button during mouse-movement the element resize behaviour will be changed depending on the element type. Lines will be changed to horizontal or vertical alignment. Rectangles and ellipses will become squares or circles according to plot coordinates (x/y-space) and not pixel coordinates. To avoid confusion with plot aspect, a button for fixed axis aspect ratio (“1:1”) was added to the plot bar.

6.6.2 Script based pick and plot from / to a Qwt-widget

To allow more complex user interaction with scripts, e.g. script based element picking, the plot functionality can be started by script either blocking or non-blocking.

```
myImage = dataObject.randn([200, 200], 'float32')
[number, handle] = plot(myImage, "itom2dQwtPlot")

# Blocking access which return the values for a single point in myElement
# Structure will be dataObject([8, 1], 'float32') with [[idx], [type], [x], [y], [0], [0], [0], [0]]
myElement = dataObject()
handle.drawAndPickElements(101, myElement, 1)

# None blocking plot
# Structure will be dataObject([1, 11], 'float32') with [idx, type, x, y, 0, 0, 0, 0]
handle.call("userInteractionStart", 101, True, 1)

# --> Read out later after plot is finished
myGeometry = handle["geometricElements"]
```

The blocking code will wait until the selection is done or the selection was aborted by user and will then return the corresponding object. The non-blocking code will return directly. To access the geometric elements the corresponding “signal” for userInteractionDone should be used to noticed the end of the user interaction.

The geometric elements can also be set by script by calling the corresponding slot.

```
myImage = dataObject.randN([200, 200], 'float32')
[number, handle] = plot(myImage, "itom2dQwtPlot")

# Add the marker to the plot
# marker is filled according to marker style definition
marker = dataObject([8,1], 'float32', data = [101.0, 1.0, 5.0, 6.0, 0.0, 0.0, 0.0, 0,0])
handle.call("plotMarkers", marker, "b", "")

# Delete all marker and than plot new marker
# marker is filled according to marker style definition
myGeometry = dataObject([1,11], 'float32', data = [101.0, 1.0, 5.0, 6.0, 0.0, 0.0, 0.0, 0,0, 0.0, 0,0, 0.0, 0,0])
handle["geometricElements"] = myGeometry
```

The geometric elements can be read any time using the property “geometricElements”.

```
# Reading geometric elements
myGeometry = handle["geometricElements"]
```

The object “myGeometry” consists of all geometric elements with in the plot. Each row corresponds to one geometric element while the parameters for each element are align column-wise. This kind of reading differs from the blocking-variant (“drawAndPickElements”). The blocking-variant returns only the data created during the current function call and ignores old geometric elements. In this case the elements are aligned column-wise. This means each column corresponds to on element while its data is stored along the rows. For the differernt definitions of the geometric elements see section “Indexing of Geometric Elements”.

6.6.3 Implemented Functions, Signals and Slots

The Qwt-plot widgets functions had to be updated. The Qwt-Widgets got the following properties, respectively setter- / getter-functions related to plotting:

- “geometricElementsCount”, get the number of geometric elements in this plot, READONLY
- “keepAspectRatio”, enable or disable a fixed aspect ratio of 1:1 for the plot canvas.
- “enablePlotting”, enable and disable plotting toolbar
- “selectedGeometry”, set / get the current selected geometric element

In complete theses the functionality of the drawing interface, the following slots have been added to the widgets:

- “plotMarkers” Add markers and geometric elements to plot according to dataObject and style type. For geometric “style is not used”, type in “b”.
- “userInteractionStart” Start a non-blocking user interaction for “multipointpick” or geometric elements with element count.
- “clearGeometricElements” Delete all existing geometric elements within the plot

To register changes in the plot elements and finished user interactions, the following signals where implemented:

- “userInteractionDone”, Emitted, when user interaction is finished or aborted
- “plotItemChanged”, Emitted, when a geometric element was changed and changing is finished. Not fully Python-Compatible.
- “plotItemDeleted”, Emitted, when a specific plot item is deleted
- “plotItemsDeleted”, Emitted, when all plot items are deleted
- “plotItemsFinished”, Emitted, when the plotting function is finished, similar to userInteractionDone

For the blocking connection the plotItem-class got the additional function drawAndPickElements(type, dataObject, count), see [plotItem](#).

Note: If the plot is embedded in a graphical user interface, the python based access to the plot via its object name

returns an instance of `uiItem`. However, you can cast this instance to `plotItem` using a python cast operator:

```
import itom
plot = itom.plotItem(myGui.plotObjectName)
```

If `myGui.plotObjectName` is not an instance of `plotItem` a runtime error is thrown. This cast is only available for itom > 1.4.0.

6.6.4 Indexing for Geometric Elements

The definition of the geometric elements depends on the implementation. The “plotMarker” and its corresponding getter- / setter-function uses a Matlab orientated structure. The structure a `dataObject` with 8 rows and `n` columns where `n` depends on the number of elements. Points are defined by their location, while ellipses and rectangles are defined by their diagonal edges.

The “geometricElements”-property uses geometric elements in a more mathematical orientated description. The `dataObject` structure is defined as “float32” with `n` by 11 elements. Most setter functions also support “float32” elements. Each of the `n` rows corresponds to an elements (except polygon-shapes). The indexing follows the `geometricPrimitive`-struct in c++.

The `geometricPrimitive` is a struct within the c-Structur of the program designed for exchanging the geometric elements from plots to other elements. The structure can be used row-wise as `dataObject` or `float32`-lists

At the moment only `tPoint`, `tLine`, `tEllipse` and `tRectangle` are supported.

The cells contain:

1. The unique index of the current primitive, castable to `int32` with a maximum up to 16bit index values
2. Type flag 0000FFFF and further flags e.g. read&write only FFFF0000
3. First coordinate with x value
4. First coordinate with y value
5. First coordinate with z value

All other values depends on the primitive type and may change between each type.

- A point is defined as `idx, flags, centerX0, centerY0, centerZ0`
- A line is defined as `idx, flags, x0, y0, z0, x1, y1, z1`
- A ellipse is defined as `idx, flags, centerX, centerY, centerZ, r1, r2`
- A circle is defined as `idx, flags, centerX, centerY, centerZ, r`
- A rectangle is defined as `idx, flags, x0, y0, z0, x1, y1, z1, alpha`
- A square is defined as `idx, flags, centerX, centerY, centerZ, a, alpha`
- A polygon is defined as `idx, flags, posX, posY, posZ, directionX, directionY, directionZ, idx, numIdx`

class ito::PrimitiveContainer

This is a container to store geometric primitives. The enum `tPrimitive` of this file defines the geometric primitives for all plots.

Author

Wolfram Lyda, twip optical solutions GmbH, Stuttgart

Date

12.2013

Public Type

tPrimitive enum

Discribes the different primitive types

See

itom1DQwtPlot, itom2DQwtPlot

Values:

- `tNoType = 0` -
- `tMultiPointPick = 6` - ! NoType for pick
- `tPoint = 101` - ! Multi point pick
- `tLine = 102` - ! Element is tPoint or order to pick points
- `tRectangle = 103` - ! Element is tLine or order to pick lines
- `tSquare = 104` - ! Element is tRectangle or order to pick rectangles
- `tEllipse = 105` - ! Element is tSquare or order to pick squares
- `tCircle = 106` - ! Element is tEllipse or order to pick ellipses
- `tPolygon = 110` - ! Element is tCircle or order to pick circles
- `tMoveLock = 0x00010000` - ! Element is tPolygon or order to pick polygon
- `tRotateLock = 0x00020000` - ! Element is readOnly
- `tResizeLock = 0x00040000` - ! Element can not be moved
- `tTypeMask = 0x0000FFFF` - ! Element can not be moved
- `tFlagMask = 0xFFFF0000` - ! Mask for the type space

6.6.5 Evaluation of Geometric Elements

The evaluateGeomtrics-widget is designed to load geometric definition stored in a float32 dataObject with a column-size of >10 elements and a row for each geometric element to display. Further more it allows the evaluation of geometric relations between the geometric primitives. See section *Custom Designer Widgets* for the widget description.

6.6.6 Demo Scripts and Examples

- **uiMeasureToolMain.py**

Description: Advanced GUI which enables geometric plotting and measurements within a 2D-QWT-Plot. This file shows how to auto-connect to signals and how to use buttons. The corresponding ui-file is uiMeasureToolMain.ui.

- **demoPickPointsAndMarkers.py**

Description: Demo for picking & plotting points and picking & plotting ellipses.

6.7 Matplotlib

itom is not only able to plot array-like structures like `itom.dataObject` or Numpy arrays but it also provides a designer plugin widget that can be used to render the output of the famous Python package **Matplotlib** into a plot window. It is even possible to integrate this type of widget, like all other designer plugins as well, into an user-defined graphical user interface.

For more information about the Matplotlib functionality, see this [this section](#).

There are further custom widgets for the QtDesigner which realized itom specific non-plotting functions. See section [Custom Designer Widgets](#) for the widget description.

EXTENDING THE USER INTERFACE OF ITOM

7.1 Customize the menu and toolbars of itom

In this section, it is shown how you can add your user-defined toolbars and menus to the main window of **itom**. Clicks to these execute arbitrary python code or methods. The creation of the toolbars, buttons and menus is done using python code, too.

7.1.1 Add toolbars and buttons

Using the embedded scripting language in **itom**, you can add your own toolbars and buttons in order to automatically execute specific **Python**-commands or -methods. Every button that is created is related to a toolbar defined by its toolbar-name. If a toolbar-name does not already exist, a new toolbar with this name is created and the button is added to this toolbar. Every toolbar can be arbitrarily positioned in **itom** or undocked to any floating position on your screen.

A single button in a toolbar is created using the command `addButton()`, vice-versa the button is removed with `removeButton()`. The syntax for both is:

```
buttonHandle = addButton(toolbarName, buttonName, code [, icon, argtuple])
removeButton(toolbarName, buttonName)
#or
removeButton(buttonHandle)
```

Your button is accessed by its name **buttonName**, which also is printed on the button if no icon is defined. The toolbar, where the button is displayed, is defined by its name **toolbarName**. You can either provide a python code snippet as string or a reference to any method or function. If this method requires parameters, you can add these parameters as tuple to the keyword-parameter *argtuple*. Additionally, it is possible to assign an icon to the button. Therefore you give the absolute path of the icon to the parameter *icon*. See the *icon section* below about the different possibilities how to assign a valid filename.

In the following example, three different buttons linking to both python code and some functions are created:

```
def test1():
    print("The button test1 has been clicked")

def test2(a,b):
    print("The result of a+b is",a+b)

addButton("myToolbar", "HelloWorld", "print('Hello World')")
addButton("myToolbar", "BtnTest1", test1, ":/application/icons/itomicon/curAppIcon.png")
addButton("myToolbar", "BtnTest2", test2, argtuple=(4,7))
```

Note: In this example, it is assumed that the module *itom* has been globally and totally imported by:

```
from itom import *
```

If this is not the case, you can also import *itom* and call *itom.addButton(...)*. Additionally, if you create a button in a toolbar, whose name already exists, the previous button will be deleted first. If you assign a code-snippet to

the `code`-parameter that includes quotation marks, make sure that the quotation mark around the code-snippet is different to the quotation marks within the code snippet or escape your inner quotation marks by a backslash.

```
addButton("myToolbar", "QuotExample", "print('different quotation mark')")
addButton("myToolbar", "QuotExample", "print(\"escaped quotation mark\")")
```

All these buttons are removed by the following lines of code:

```
removeButton("myToolbar", "HelloWorld")
removeButton("myToolbar", "BtnTest1")
removeButton("myToolbar", "BtnTest2")
```

As an alternative approach to delete a button, use its handle, returned by the `addButton` method and pass it to `removeButton`. Its advantage is, that exactly the button, that has been created is deleted and not a button with the same name that has been created by another instance and is for instance connected with another code snippet.

```
handle = addButton("myToolbar", "HelloWorld", "print('Hello World')")
try:
    removeButton(handle)
except RuntimeError:
    #button did not exist any more (e.g. has been overwritten
    #by another addButton command with the same button and toolbar name)
    pass
```

Note: If the last button of a toolbar has been removed, the toolbar is removed as well.

7.1.2 Create menus

You can not only add buttons to the toolbar of **itom**'s main window but also create your menu and sub-menu structure. Therefore the commands `removeMenu()` are available.

There are three types of menu items, that can be created:

- **MENU** (*itom.MENU* [2]) creates a menu-item, having any possible sub-item. This item cannot be connected to any code. Every menu always starts with an item of this type.
- **BUTTON** (*itom.BUTTON* [0]) creates a real menu-item, that is the mode of the menu-tree. Only a click to these items can execute code.
- **SEPARATOR** (*itom.SEPARATOR* [1]) creates a separator-item in the menu or submenu. It is also not connected to any code.

Any menu-item is defined by its key. The key is a slash-separated list, where the single items stand for the path one has to walk through the menu-tree in order to access the desired item. If an item with a complex-tree structure is created where some of the parent-nodes do not already exist, they are iteratively created (type **MENU**) in order to finally create the desired node-element.

The call to `addMenu()` is as follows:

```
addMenu(type, key [,name, code, icon, argtuple])
```

Hereby *type* is the item's type like stated above and *key* denotes the absolute key to the item, which also indicates the tree-structure, where the item should be added. The name of your item is given by *name*, while *code*, *icon* and *argtuple* have the same meaning like in the case of adding a button to the toolbar (*see above*).

Here are some examples for creating a menu:

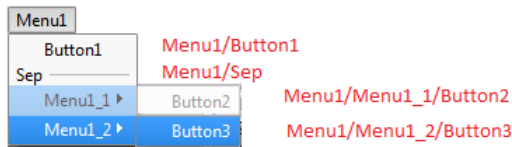
```
1 def btn2(arg=None):
2     print("button 2 clicked:",arg)
3
4 addMenu(itom.BUTTON, "Menu1/Button1", "Button1", "print('button1 pressed')")
5 addMenu(itom.SEPARATOR, "Menu1/Sep")
6 addMenu(itom.MENU, "Menu1/Menu1_1")
```

```

7 addMenu(itom.BUTTON, "Menu1/Menu1_1/Button2", "Button2", btn2)
8 addMenu(itom.BUTTON, "Menu1/Menu1_2/Button3", "Button3", btn2, argtuple=[2])

```

In line 4, the *Button1* is created. Since its parent node *Menu1* does not exist, it will be created (type *MENU*). Next, in line 5 a separator is added as subitem of *Menu1*. Hence, it is appended to *Button1*. Afterwards two sub-nodes *Menu1_1* and *Menu1_2* are added which both have a children respectively, called *Button2* and *Button3*.



Note: Please consider, that in line 8 an *argtuple* is appended to the function-call to *btn2*. Although only one argument is passed it must be included in a tuple. Usually a tuple is created by the bace operator (2). However, since only one argument is given, python is interpreting this brace-operator as mathematical expression and it is reduced to 2 only. Therefore, we use the square bracket in order to create a list, that is implicitly converted to a tuple.

In order to remove any menu item including its subitem, call `removeMenu()` with the key-word of the specific item. For instance, the removal of all menus created above, is done by:

```
removeMenu("Menu1")
```

The only argument of the command `removeMenu()` is only the key of the menu-item to delete.

7.1.3 Icons in user-defined toolbars and menus

Both for toolbar-buttons as well for menu-entries you can assign an arbitrary icon. Usually it is recommended to have an icon file with a size of 24x24 Px or below (will be automatically resized) in any image format (*png* recommended, available is *bmp*, *gif*, *jpg*, *tiff*...). The argument **icon** of the commands `addButton()` and `addMenu()` must be a string with the absolute or relative filename of the icon-file. The relative filename is always considered to be relative with respect to the current working directory, as it is printed at the right bottom side of **itom**'s main window (or use the command `getCurrentPath()`). Besides that, you can also pass an absolute path to your icon-file. The command `getAppPath()` returns the absolute path of the **itom**-application. Additionally you can use methods from the **Python**-module `os.path` in order to create valid absolute paths.

Besides assigning an external icon-file to the **icon** parameter, **itom** also gives you access to any icons that are compiled as resources within the **itom** application. All icons, that are included in these resources are listed in the **icon browser**:

The icon browser is accessible in any script window by its menu **edit >> icon browser** (or Ctrl+B). If you found your desired icon, double click on the entry in order to copy the appropriate string to the clipboard and paste it afterwards into your script. Resource locations always start witha colon (:) sign.

7.2 Show messages, input boxes and default dialogs

itom gives you the possibility to show the standard message and input boxes as well as standard dialogs per **Python** script command.

7.2.1 Message boxes

There are three types of message boxes that you can show via a script command. These are divided into the four types **Information**, **Warning**, **Question** and **Critical**.

The corresponding commands are the following static methods in the class `ui`:

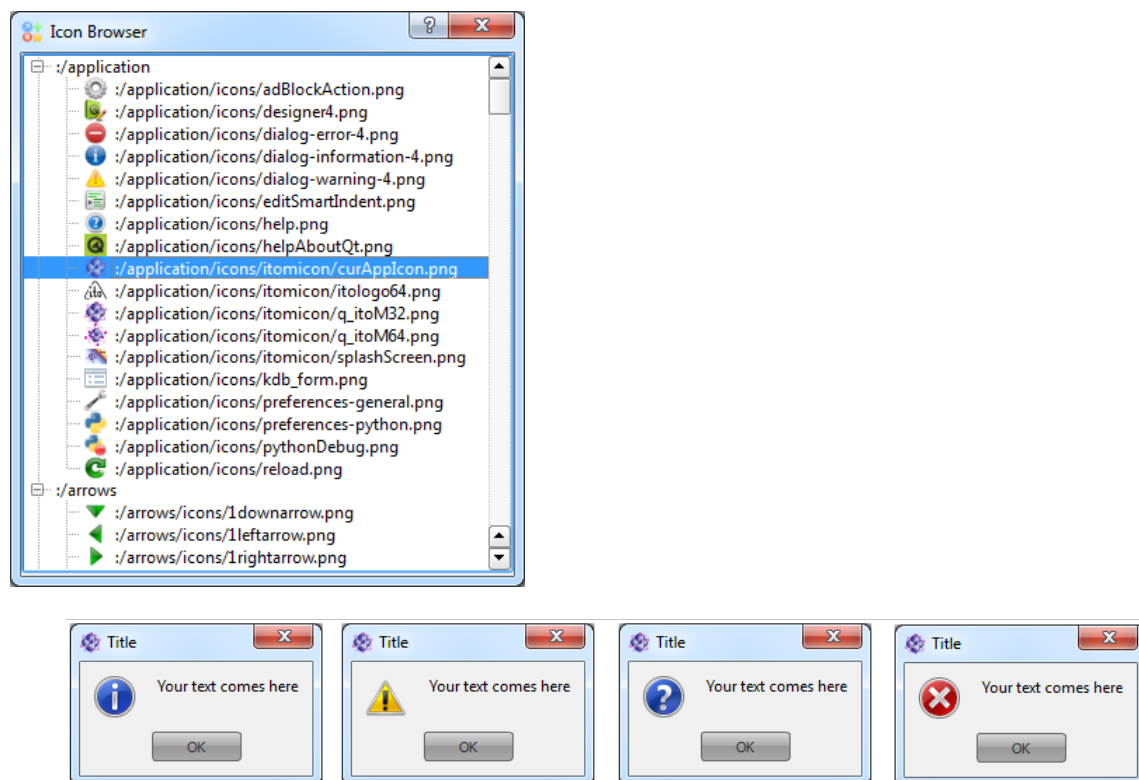


Fig. 7.1: Appearance of the different message boxes on a Windows 7 operating system. From left to right: *Information*, *Warning*, *Question*, *Critical*

- `msgInformation()`
- `msgWarning()`
- `msgQuestion()`
- `msgCritical()`

All these methods have the same syntax:

```
(button,buttonText) = msgInformation(title, text [,buttons, defaultButton, parent])
```

The parameters *title* and *text* are the strings for the titlebar and the content area of the message box respectively. The other three parameters are optional: *buttons* is an or-combination of button-IDs, denoting the different buttons you want to display at the bottom of the message box. The possible values correspond to the enumeration `QMessageBox::StandardButton` of the *Qt*-framework and are:

Enumeration	Value
ui.MsgBoxOk	1024
ui.MsgBoxOpen	8192
ui.MsgBoxSave	2048
ui.MsgBoxCancel	4194304
ui.MsgBoxClose	2097152
ui.MsgBoxDiscard	8388608
ui.MsgBoxApply	33554432
ui.MsgBoxReset	67108864
ui.MsgBoxRestoreDefaults	134217728
ui.MsgBoxHelp	16777216
ui.MsgBoxSaveAll	4096
ui.MsgBoxYes	16384
ui.MsgBoxYesToAll	32768
ui.MsgBoxNo	65536
ui.MsgBoxNoToAll	131072
ui.MsgBoxAbort	262144
ui.MsgBoxRetry	524288
ui.MsgBoxIgnore	1048576
ui.MsgBoxNoButton	0

For an **or**-combination, use the bar-operator (`|`). The parameter *defaultButton* only accepts one button value of buttons, listed in the parameter *button*. The specific button then becomes the default button that is selectable by simply pressing the return-key. Additionally, you can let the message box be a child of any other instance of class *ui*. This is useful, if you want to make the message box modal with respect to any *user defined dialog* and not with respect to the main window of **itom**.

The return value is a tuple containing two values. The first is the value of the button that has been pressed to close the message box with respect to the enumeration values stated above. The second value is the text of the pressed button (e.g. “Ok”).

An example for a question message box is:

```
(btn, btnText) = ui.msgQuestion("Silly question", "Do you like itom?", \
    ui.MsgBoxYes | ui.MsgBoxNo, ui.MsgBoxYes)
print("The user pressed the button", btnText, "with ID", btn)
if (btn == ui.MsgBoxNo):
    print("what can we do in order to change your opinion?")
```

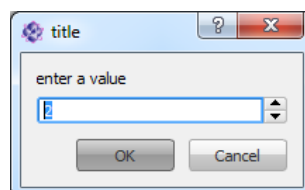
7.2.2 Input boxes

The class *ui* also contains another group of static methods, that can be used in order to show modal input boxes. These are:

Integer input box

Use *getInt()* in order to ask the user for a fixed-point number:

```
(value, accepted) = ui.getInt(title, label, defaultValue, min, max, step=1)
```



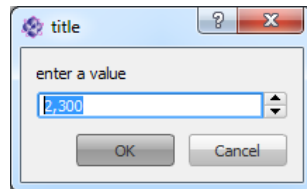
The minimum and maximum value of the input box is set to the values *min* and *max* or to the full integer range, if not otherwise stated. The step-size *step* defines the iteration value when the arrows are clicked. The return value is

a tuple that contains the current value of the input box and **True** if the dialog has been accepted, otherwise **False**, the value still contains the last value of the spinbox.

Double input box

Use `getDouble()` in order to ask the user for a double-precision floating-point number:

```
(value, accepted) = ui.getDouble(title, label, defaultValue, min, max, decimals=1)
```

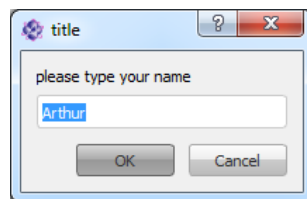


The minimum and maximum value of the input box is set to the values *min* and *max* or to the full double range, if not otherwise stated. In this case you can assign the number of decimals that are shown (default:1). The step size is always set to 1. The return tuple has the same form than for the integer input box.

Text input box

Use `getText()` in order to ask the user for a string:

```
(value, accepted) = ui.getText(title, label, defaultValue)
```

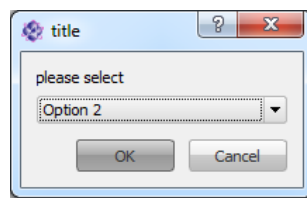


The default text can be passed as parameter *defaultValue*. The return tuple also returns the current value of the textbox and the boolean variable **True** if the dialog has been accepted, else **False**.

Option selection dialog

Use `getItem()` in order to let the user select an item from a given list of items:

```
(value, accepted) = ui.getItem(title, label, stringList [,currentIndex=0, editable=False])
```



Use the parameter *stringList* in order to pass any sequence (tuple or list) of strings that are displayed as options, the user can choose. You can preselect any item by setting the parameter *currentIndex* to the item's index (zero-based, default:0). Additionally it is possible to keep the selection dialog editable, hence, the user can also type its own string (if **True**, default: **False**).

The returning tuple again contains the current active or typed string value and the boolean variable, that is only **True** if the dialog has been accepted.

7.2.3 Standard dialogs

itom provides access to a standard dialog to indicate an existing directory, to choose an existing file or to indicate a path for saving a file.

Indicate existing directory

The dialog is wrapped by the static method `getExistingDirectory()`:

```
ret = ui.getExistingDirectory(caption, startDirectory [, options = 1, parent])
```

The parameters are as follows:

- **caption** is the title of the dialog
- **startDirectory** is a string containing an absolute path to the default directory
- **options** is an optional or-combination of one of the following options:
 1. ShowDirsOnly [default], only directories are shown in the dialog
 2. DontResolveSymlinks, if indicated symbolic links are not shown
 other values can be taken from the Qt-enumeration **QFileDialog::Option**
- **parent** can be an instance of *ui*, such that the dialog becomes modal and on top of this user defined dialog

The return value **ret** of this dialog is either **None** if the dialog has been rejected or else the absolute path to the chosen directory.

OpenFileName-Dialog

Use the static method `getOpenFileName()` to show this dialog. By this dialog, the user can select one specific existing file in any directory. This is usually taken for opening files.

```
ret = ui.getOpenFileName([caption="", startDirectory="", filters="", \
    selectedFilterIndex=0, options=0, parent])
```

The title of the dialog can be assigned by the parameter *caption*. You can choose a default directory by the string *startDirectory*. If this is not given or empty, the current working directory is taken. The possible file filters are set by the string *filters*. This is a double-semicolon separated list of entries. An example is:

```
Images (*.png *.jpg)
Text Files (*.txt);;Itom Data Collection (*.idc);;Scripts (*.py *.pyc)
```

The selected, default index is given by *selectedFilterIndex*, where 0 selects the first item in *filters*. You can also pass some additional options to the dialog by the parameter *options*. This basically is an or-combination of the Qt-enumeration **QFileDialog::Option**. Reasonable values are:

- 0x02, DontResolveSymlinks. If given, symbolic links are not shown in the dialog
- 0x40, HideNameFilterDetails. This only shows the names of the filters, but not the pattern mask.

The function returns *None* if the dialog has been rejected, else it is the absolute filename of the selected file.

SaveFileName-Dialog

Use the static method `getSaveFileName()` to show a dialog, where the user can select a new filename or an existing file in any directory. This is usually taken for saving files.

```
ret = ui.getSaveFileName([caption, startDirectory="", filters="", \
    selectedFilterIndex=0, options=0, parent])
```

The parameters are the same than for the open-file-dialog above. However for options, reasonable values are now:

- 0x02, DontResolveSymlinks. If given, symbolic links are not shown in the dialog
- 0x04, DontConfirmOverwrite. If given, the user don't has to confirm if an existing file has been choosen.
- 0x40, HideNameFilterDetails. This only shows the names of the filters, but not the pattern mask.

For combining different options, use the or-operator |.

7.3 Creating advanced dialogs and windows

With **itom** it is not only possible to add menus and toolbar elements to the main GUI of **itom** or to use the default set of input and message boxes, but it is also possible to create own user interfaces. These interfaces are designed by help of a WYSIWYG (“what you see is what you get”) design tool (Qt Designer). The logic behind the surfaces is then scripted using **Python**. Therefore it is possible to change the appearance of control elements at runtime or to connect a signal, emitted when for instance clicking on a button, with a user-defined python method.

In this chapter, the creation of such user interfaces is explained.

7.3.1 Qt Designer

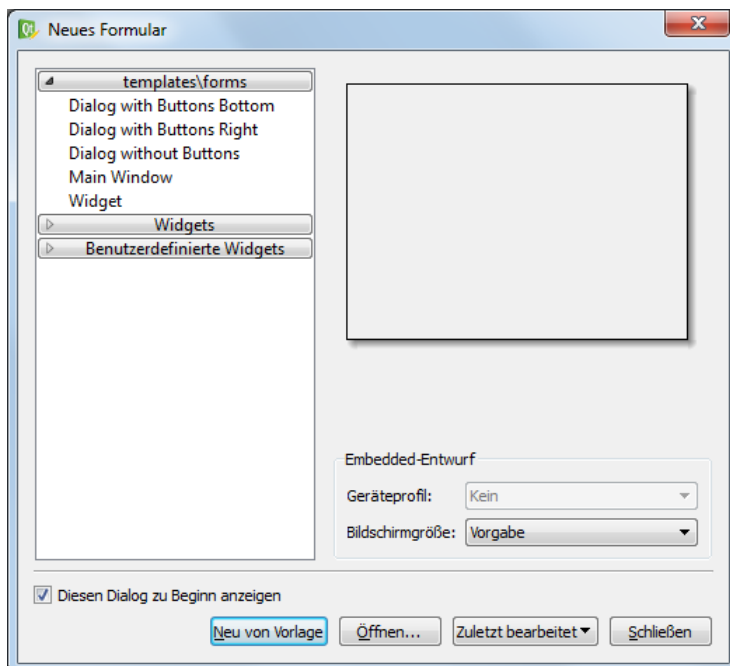
The Qt Designer can be used to create a GUI for interaction with the **itom** software.

For details see the Qt Designer documentation under <http://qt-project.org/doc/qt-4.8/designer-manual.html>

In order to start the **Qt Designer**, click on the corresponding icon in the toolbar of **itom**:



or double-click on a corresponding **ui**-file in the file system widget of **itom**. In the first case, **Qt Designer** shows an initialization dialog, where you can choose the base type of the user interface you want to create.



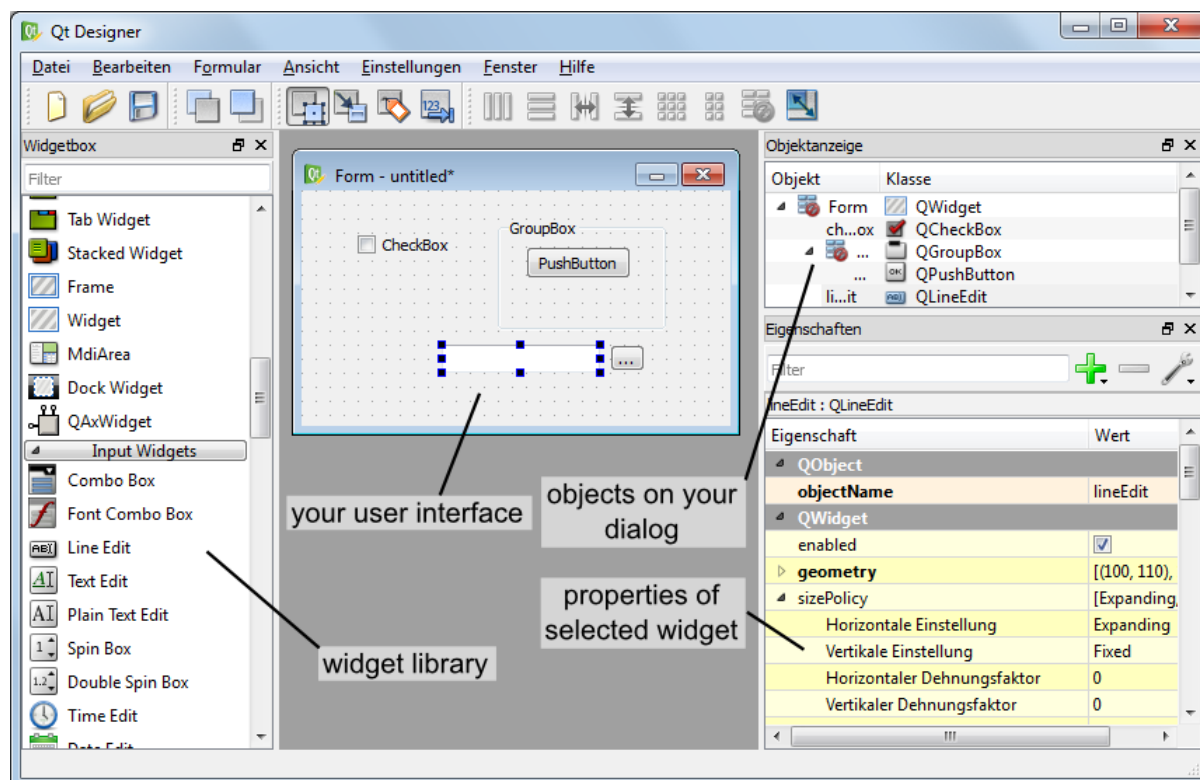
Note: There is a known issue in the setup 1.4.0 (or earlier) concerning an unsuccessful startup of the external Qt

designer. If you want to open the designer using the button in the toolbox of the itom main window, the startup may fail. This bug is known and will be fixed in future releases. Until then, please open the designer either by starting the **designer.exe** in the application folder of itom or open an existing ui-file (e.g. in the demo folder of itom). This issue only affects setup versions (32bit and 64bit) for Windows.

In principle you have the possibility to choose between three different base layouts:

1. **Dialog.** A Dialog is usually displayed on top of the main window and only has got one close-button in its title bar. Often, dialogs are used for configuration dialogs where the user finally closes the dialog using one of the standard buttons (OK, Cancel, Apply...) in order to confirm or reject the current changes in the dialog. A dialog cannot have its own toolbar, menu or status bar.
2. **Main Window.** A main window is a fully equipped main window, which can be minimized, maximized, can have toolbars, menus and a status bar. Therefore it is recommended to use this type of user interface for the main window of your measurement system. Like a dialog, it is possible to show the main window on top of **itom** (as sub-window of **itom**) or as independent window, which has its own icon in the windows tray.
3. **Widget.** A widget is the base class for all control elements provided by **Qt**. Therefore a widget does not have any title bar or windows frame. Nevertheless you can choose a widget for your user interface, since **itom** provides the possibility to stack this widget into a default dialog which can optionally show some default buttons on the right side or at the bottom of the dialog. This is the easiest way the generate a configuration dialog in **itom**, since you do not need to script the necessary methods handling clicks on one of these buttons. In this case, **itom** automatically gets full information about the close status and type of closing of the dialog (accepted, rejected...).

After having chosen one of these base layouts (types), your surface is displayed in the middle of the **Qt Designer** and you can start to drag elements from the widget library on your surface. If the **Qt Designer** is started from **itom** you will even find a section **ITOM Plugins** in the library list, which contains all loadable designer plugins that are provided by **itom** and can also be placed on your surface. The choice of these plugins depend on the designer plugins that are currently available in your installation of **itom**.



After having placed one widget on the canvas, you will see its properties in the property toolbox of **Qt Designer**. Every widget has the common property **objectName**. If you assign a unique object name to any of your control elements, it is possible to access and manipulate this widget from a **Python** script in **itom** using this name, too. In general many of the properties that are visible in the property toolbox can afterwards be read or changed by an appropriate script (depending on the data type of the property).

The alignment of control elements on the surface is mainly controlled by so-called layout elements. These layouts together with size policies that can be assigned to every widget control the appearance of the entire user interface and provide the feature that the dialog can be changed in size whereas all widgets are dynamically repositioned. For more information about laying out your user interface, see <http://qt-project.org/doc/qt-4.8/designer-layouts.html>.

Finally, save your user interface under a convenient filename with the suffix **.ui**.

Widget Library

In principle, you are allowed to place every widget on your user interface that is available in the widget library (widget box) of **Qt Designer**. Later, you will learn how you can access properties of any widget (read and/or write) and how you can call specific functions provided by any widget. However, you will also learn that you do not have access using **Python** to all functions a widget has and you are not able to sub-class any widget, like you can it using a native **C++** program. Therefore, it is not recommended to place any widget from the group **Item Views (Model-based)** on your user interface since only few functions of these widgets are accessible by a **Python** script. If you need a list box, use the item-based list widget. **itom** also provides some widgets (section **ITOM widgets**) that can be placed on your user interfaces, for instance some plot widgets or the widget for plotting the result of the python module *matplotlib* (see *Python-Module matplotlib*).

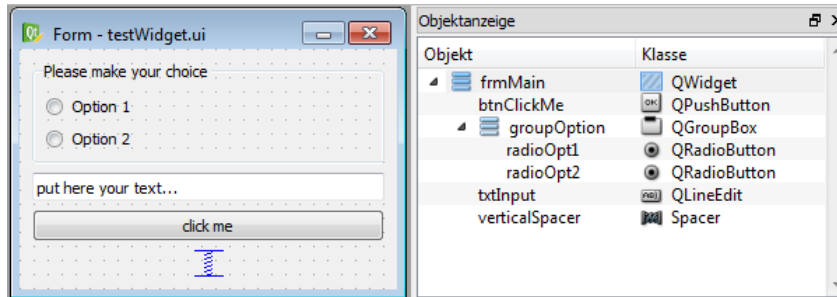
7.3.2 Loading user interface in itom

In this section, an introduction is given how to create and load user interfaces in **itom** depending on different type-attributes.

Widget embedded in itom-dialog (TYPEDIALOG)

Like described above, the easiest and most comfortable way to load user interfaces in **itom** is to use the type **TYPEDIALOG**. In **Qt Creator** you design a widget with your individual content and then when loading this GUI in **itom**, the widget is embedded in a dialog provided by **itom**, which optionally adds a horizontal or vertical button bar at the right side or at the bottom of the dialog.

Let us create an exemplary user interface. In **Qt Creator** the following widget has been created:



On the right side of the widget *testWidget* you see the hierarchical organization of objects that are put on the widget. At first, a group box has been placed on the widget. Inside of this group box two radio buttons have been placed using a simple drag&drop from the widget library. Both radio buttons are aligned inside of the group box with a vertical layout. This is reached by a right-click on the group box and choosing *vertical layout* from the *layout* menu. Below the group box, a widget of type *lineEdit* and a push button (type *pushbutton*) have been placed. Finally the three main elements are also aligned in a vertical layout with respect to the overall widget. This can be achieved by a right click on an empty space of the widget or directly in the *object inspector*. If you increase now the size of the overall widget, you will see that all sub-elements are resized according to their layout. Since we don't want sub-widgets to be vertically stretched and distributed, a vertical spacer element has been placed at the bottom of the vertical layout stack.

The following properties have been directly set in **Qt Creator**:

- group box: *objectName*: groupOption, *title*: 'Please make your choice'
- push button: *objectName*: btnClickMe, *text*: 'click me'
- line edit: *objectName*: txtInput, *text*: 'put here your text...'
- radio buttons: *objectName*: radioOpt1 and radioOpt2, *text*: 'Option 1' and 'Option 2'

The entire widget is saved under the filename *testWidget.ui* in an arbitrary directory.

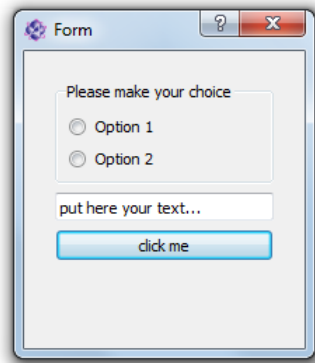
Then you can load and show the widget in **itom** by creating a python-script in the same directory with the following content. You can also directly type these lines into the command line of **itom**, however, you should then assure that the current directory is equal to the directory where the user interface has been stored.

```
dialog = ui("testWidget.ui", ui.TYPEDIALOG) #loading dialog
result = dialog.show(1) #modally show, wait until the dialog has been closed
print("The dialog has been closed with code", result)
```

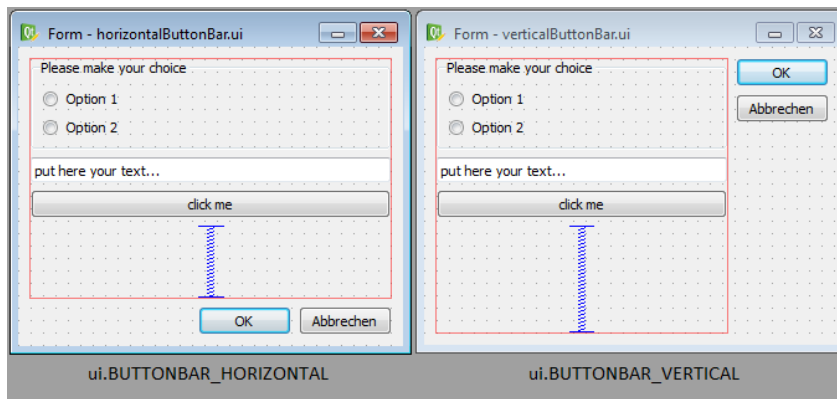
At first, an instance of class *itom.ui* is created that is given the name of the user interface file. This instance can then be accessed by the name *dialog*. By calling the method *show()*, the dialog is shown. Since the parameter has been set to **1**, the dialog is shown in a modal style, such that python waits until the dialog has been closed again and **itom** is entirely blocked during that time. However, then it is possible to get informed about the way the dialog is closed, such that the variable *result* will be set to **0** if the user closed the dialog using a cancel button (not available here) or the close button in the title bar or **1** if the user clicked an **OK**-button.

It is also possible to open the dialog in a non-modal version or to open it in a modal style however to immediately force python to continue the script execution. This depends on the parameters of *show()*. However only in the modal case above, the closing result can be tracked by **Python**. Additionally, this is also only possible if a widget is embedded in a dialog, given by **itom**, like it is always the case if you create an instance of *itom.ui* with the second parameter set to **ui.TYPEDIALOG**.

```
>>dialog = ui("testWidget.ui", ui.TYPEDIALOG)
result=dialog.show(1)
```



Right now, you don't have the possibility to quit the dialog using any button (**OK**, **Cancel**...). In order to obtain a button bar with these buttons, the call to the class `itom.ui` needs to be changed. There is the choice between two different appearances of a button bar, which can be automatically added to your widget:



Next, you need to select which buttons should be included in the button bar. This is done by creating a python dictionary, where each element corresponds to one button. The key-word of the item corresponds to the role of the button (see enumeration `QDialogButtonBox::ButtonRole`* of the **Qt**-library documentation) and the value is the text of the button. Common roles are:

- “AcceptRole”: Use this role for an **OK**-button. The dialog is closed and the return value in modal style is 1.
- “RejectRole”: Use this role for a **Cancel**-button. The dialog is also closed but the return value is 0.

Finally, the call to `itom.ui` must be in the following way, in order to get an auto-generated button bar:

```
dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_VERTICAL, \
    {"AcceptRole": "OK", "RejectRole": "Cancel"})
#or
dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_HORIZONTAL, \
    {"AcceptRole": "Yes", "RejectRole": "No"})
```

Note: You can also use a keyword-based call to `ui` since every parameter has its default value such that you can omit parameters beside the first one. For more details about all parameters, keywords and its default values see `itom.ui`.

The dialog is closed and deleted if the variable **dialog** is deleted using the command `del`.

Main window or dialog (TYPEWINDOW)

If you are not interested in the exact return value of the dialog but you want to have full control and all available functionalities of any dialog or main window, create an user interface based on a **dialog** or **main window** in **Qt Designer**.

The figure shows an exemplary user interface (**testWindow.ui**) that is based on a main window. On the right side, there have been added three buttons, nested in a vertical layout. On the left side, there is a list widget (objectName: **listWidget**, type: **List Widget**). Additionally a menu has been added that consists of three items.

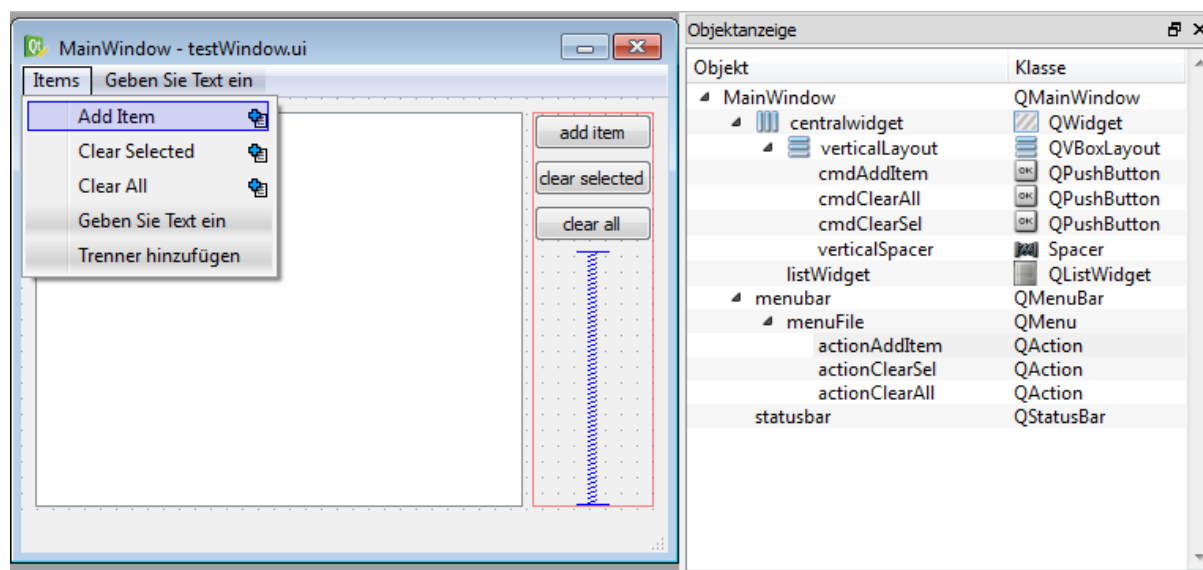
This main window can now be shown using the following code snippet:

```
win = ui("testWindow.ui", ui.TYPEWINDOW)
win.show() #this is equal to win.show(0) -> non-modal
```

Then, the window is shown on top of the main window of **itom**, since it is considered to be a child of **itom**. If you don't want this, you can also add the keyword-parameter `childOfMainWindow=False` to the call of `itom.ui`:

```
win = ui("testWindow.ui", ui.TYPEWINDOW, childOfMainWindow=False)
```

Here, you need to use the keyword, since the parameters `dialogButtonBar` and `dialogButtons` (used for `TYPEDIALOG`) are not given in this case, since they are useless in case of `TYPEWINDOW`. If your window is no child of **itom**, it gets its own icon in the Windows tray bar and does not stay on top of **itom**.



Main window or widget as dockable toolbox (TYPEDOCKWIDGET)

It is also possible to integrate user-defined main windows or widgets as dockable toolbox in the main window of **itom**. This is done using the type value **ui.TYPEDOCKWIDGET**. Then the widget is registered as dockwidget in the main window of **itom** and once it becomes visible, its startup position is at the top-center position. You can define the initial dock area using the optional argument *dockWidgetArea* of the class *itom.ui*.

```
win = ui("testWindow.ui", ui.TYPEDOCKWIDGET, dockWidgetArea = ui.RIGHTDOCKWIDGETAREA)
```

Possible values for *dockWidgetArea* are:

```
ui.LEFTDOCKWIDGETAREA    =    1    ui.RIGHTDOCKWIDGETAREA    =    2
ui.TOPDOCKWIDGETAREA    =    4    ui.BOTTOMDOCKWIDGETAREA    =    8
```

7.3.3 Accessing control elements

Until now, you know how to design an interface and how to show it using **itom**. This and the following sections explain how you dynamically interact with the user interface and its elements. One elementary tool for this is to access any desired element of the GUI. For instance, if you want to change properties of a button or the text of a linedit-widget, you first need to access these elements.

The accessing is simply done by the unique and specific **objectName** of each element and the dot-operator (.). Let's take the first example **testWidget.ui** again. The dialog has been assigned the variable **dialog**. Then have the following possibilities to access its elements:

```
elemGroup = dialog.groupOption # access the group box by its objectName

elemRadioOpt1 = dialog.radioOpt1 # OR
elemRadioOpt1 = elemGroup.radioOpt1

textfield = dialog.txtInput
```

Each variable created by the code block above is an instance of *itom.uiItem*. It is this class that defines the dot-operator. Looking at the example of accessing the first radio button, which is a child of the group-box, it is both possible to access the group button by its **objectName** as child of the entire dialog or as child of the groupbox. This is feasible since the class *ui* is derived from *uiItem*, such that the dot-operator not only works for entire dialog references but also for accessing sub-elements of other widgets. However, since each **objectName** is unique among all elements of the entire dialog, it doesn't matter how to access any element.

But why do we need to access these elements? Why do they return their own instance of class `uiItem`. These questions are answered in the following sections...

7.3.4 Getting and setting properties

As already mentioned, you can read or write most properties of any element that are also listed in the property toolbox of **Qt Designer**. Properties are also separately listed in the corresponding **Qt** documentation. In general it makes sense to set properties - when offline possible - in the **Qt Designer**. This is a little bit more efficient and keeps your script tiny. Getting and setting properties is possible if you have an object of type `uiItem`. Therefore you need to get this object like described in the section above.

Getting the property value can either be done by using the mapping-operator `[]` or by using the method `getProperty()`. For instance, if you want to get the current text and the enabled status of the textfield in dialog `testWidget.ui` from the first example, you can use one of the following possibilities:

```
#1. possibility
text = dialog.txtInput["text"]
enabled = dialog.txtInput["enabled"]

#2. possibility
[text,enabled] = dialog.txtInput.getProperty(["text", "enabled"])
```

In order to set one or multiple properties, you can use similar methods. Simply assign a value to the mapping-operator `[]` or use the method `setProperty()`.

```
#1. possibility
dialog.txtInput["text"] = "new text for this textfield"
dialog.txtInput["enabled"] = False

#2. possibility
dialog.txtInput.setProperty( {"text":"new text for this textfield", \
                             "enabled":False} )
```

If you use `setProperty()`, you always need to pass a dictionary as argument. This dictionary can contain one or multiple properties, where the keyword always is the property-name (string) and the value is the corresponding new value (type depends on corresponding C++ type). For more information about supported datatypes, that can be accessed by python in **itom** see [Supported datatypes](#).

7.3.5 Supported datatypes

The classes `itom.ui` and `itom.uiItem` are the connection between any python-script in **itom** and GUI-elements, written in C++ and provided by **Qt**. Therefore, it is necessary to transform types from python to corresponding C++-structures and vice-versa. The following table lists some convenient type casts. In general, it is always tried to convert the input type to the desired destination type, such that a number can also be transformed to a string, if it is always known, that the destination requires a string.

C++/Qt-Type	Python-Type
QString	str or any type, that has a string representation
QByteArray	unicode or byte type
QUrl	any string that can be interpreted as Url
bool	any type that can be casted to a boolean value (1,0,True,False...)
QStringList	any sequence that only contains values castable to QString
int, short, long	integer, floats are rounded to integer, True=1, False=0
unsigned int ...	integer, floats are rounded to integer, True=1, False=0
float, double	integer, floats, True=1.0, False=0.0
QVector<int>	any sequence whose values are castable to int
QVector<double>	any sequence whose values are castable to double
PCLPointCloud	<i>pointCloud</i>
PCLPoint	<i>point</i>
PCLPolygonMesh	<i>polygonMesh</i>
DataObject*	<i>dataObject</i> or any type convertible to an array (see numpy)
AddInDataIO*	<i>dataIO</i>
AddInActuator*	<i>actuator</i>
QVariant	any of the types above can be transformed to QVariant
QVariantMap	a dictionary where keys are strings and values are generally convertible.
QVariantList	any sequence whose items can be convertible.
QRegion	<i>region</i>
QColor	string with color name or hex-value or <i>rgba</i>
Enumeration	integer with value or string with key (setter only)
QTime	datetime.time object
QDate	datetime.date object
QDateTime	datetime.datetime object
QFont	<i>font</i>

If a property or other arguments in **Qt** require other datatypes, it is possible to implement a converter for them. It only becomes a little bit more difficult for pointers to extended C++ or **Qt** classes. The conversion is mainly done in the **itom** class **PythonQtConversion**.

7.3.6 Connecting signals

Now, you know how to change properties of dialogs at runtime of **itom** using a small python script snippet. In this section, you will learn how you can let **itom** a specific python-method for instance if a button on the user interface is clicked. Whenever something is changed in a user interface or the user starts to interact with the interface, any type of event is emitted. In **Qt** many of these events are specially handled and called signals. For instance, if an user clicks a button, toggles a checkbox, triggers an item in a menu or selects an item in a list widget, a signal is emitted or sent.

The counterpart to a signal is called slot. **Qt** provides the possibility to **connect** a signal with a slot, under the only condition, that both have exactly the same order and type of arguments. It is even possible to connect the same signal to various slots. Whenever a signal is emitted, all connected slots are executed. **itom** provides you the possibility to define slots in form of ordinary python methods or functions and to also connect them to signals of widgets on your user interface.

For establishing the connection, you need again a reference to the specific widget on the user interface. This reference is any variable of type *uiItem*. Next, you need the name and the arguments of the **Qt** signal, you want to connect to. This information can be obtained by the **Qt** documentation. For instance, if you need any signal that a widget of type **QPushButton** (the type of our push button, placed in the user interface in file **testWidget.ui**), go to <http://qt-project.org/doc/qt-4.8/qpushbutton.html>. Unfortunately, you won't find a headline called **Signals** at this page, since **QPushButton** does not directly declare any signal. However, you can see under **Additional Inherited Members**, that **QPushButton** inherits signals from its base classes. The most important signals are inherited from **QAbstractButton**. Click on its link and you will see the available signals for a push button:

```
void clicked ( bool checked = false )
void pressed ()
```

```
void released ()
void toggled ( bool checked )
```

If any argument provides a default value, you can also omit the specific argument. Select the signal that is convenient for you and create its string-signature. The signature always contains the following structure:

```
signature = "signalName(typeName1, typeName2, ...)"
```

For instance, the signatures for the signals above are:

```
"clicked()" or "clicked(bool)"
"pressed()"
"released()"
"toggled(bool)"
```

Then, create a python method in your script, which you want to consider to be a slot and that should be connected with the signal. This method always requires the same number of arguments than given in the signature. If you want to connect a signal to a method that is a bounded method of a class in python, the first argument **self** does not count to the number of total arguments, hence, you always need to define the first parameter **self**, like it is the case for bounded methods.

Finally, use the method `connect()` in order to establish the connection. For instance let us create a method, that should show a message when the push button “click me” on the first exemplary dialog (*testWidget.ui*) has been clicked:

```
dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_VERTICAL, \
    {"AcceptRole": "OK", "RejectRole": "Cancel"})

def showMsg():
    #slot executed in button 'click me' is clicked
    ui.msgInformation("itom", "you pressed the button click me")

#connect(signature, method)
dialog.btnClickMe.connect("clicked()", showMsg)

#show dialog
dialog.show()
```

You have seen that the method `connect()` of the element *dialog.btnClickMe* (the push button) has been called. Its first argument is the signature of the signal, as second argument the reference to the slot-methods is given. If you integrate the dialog within a class and the slot is a member of this class, too, the exemplary code can look as follows:

```
class MyDialog():

    def __init__(self):
        self.dialog = ui("testWidget.ui", ui.TYPEDIALOG, ui.BUTTONBAR_VERTICAL, \
            {"AcceptRole": "OK", "RejectRole": "Cancel"})
        self.dialog.btnClickMe.connect("clicked()", self.showMsg)
        self.dialog.show()

    def showMsg(self):
        ui.msgInformation("itom", "you pressed the button click me")

#instance of class MyDialog
test = MyDialog()
```

Let us use the second example **testWindow.ui**. If you want a python method to be executed if the user clicks an action in the menu of the main window, you should connect the signal **triggered()** of every item in the menu with your method. In **Qt** such an item is an instance of *QAction* and is also accessed by its *objectName*.

```
win = ui("testWindow.ui", ui.TYPEWINDOW)
```

```
def addItem():
    print("action addItem clicked")

win.actionAddItem.connect("triggered()", addItem)
#actionAddItem is the objectName of the action
win.show()
```

7.3.7 Calling slots

Widgets on user interfaces not only emit signals but they also have slots defined, such that you can connect other signals (e.g. from other widgets) to these slots. Using a python script in **itom** you can also call (*or*: invoke) these slots.

In order to invoke a slot, call the method `call()` of any element on your user interface. For instance, in order to clear the list widget (*objectName*: `listWidget`) of **uiWindow.ui**, you can invoke its public slot `clear()`:

```
win = ui("testWindow.ui", ui.TYPESWINDOW)
listWidget = win.listWidget
listWidget.call("clear")
```

Here, the method `call()` is only called with one argument, the name of the slot in **Qt**. If this slot would have any arguments that can be converted from **Python** (see *Supported datatypes*), add these arguments as further parameters to the call.

Unfortunately, there are some methods of important widgets in **Qt**, which are not defined to be a *public slot*. For instance, the methods to add item(s) to a list widget are no slots. However, there are some exceptions defined in **itom** such that some *public methods* of widgets can also be called with the method `call()`. These exceptions are contained in the following table:

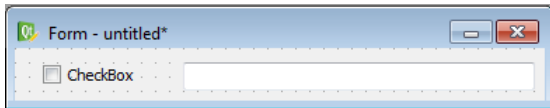
Widget / ClassName	Public Method
QWidget	void resize(int,int)
QWidget	void setGeometry(int,int,int,int)
QListWidget	void addItem(QString)
QListWidget	void addItems(QStringList)
QListWidget	void selectedRows() returns a tuple of all selected row indices
QListWidget	void selectedTexts() returns a tuple of all selected values (as strings)
QListWidget	void selectRows(QVector<int>) select the rows with the given indices (ListWidget must be in multi-selection mode)
QComboBox	void addItem(QString)
QComboBox	void addItems(QStringList)
QComboBox	void removeItem(int)
QComboBox	void setItemData(int,QVariant) sets the value of the Qt::DisplayRole (displayed text) of the item with the indicated index
QComboBox	void insertItem(int,QString)
QTabWidget	int isTabEnabled(int)
QTabWidget	void setTabEnabled(int,bool)
QMainWindow	uiItem statusBar() returns a reference to the statusbar widget
QMainWindow	uiItem centralWidget() returns a reference to the central widget of the mainWindow
QTableWidget	void setHorizontalHeaderLabels(QStringList)
QTableWidget	void setVerticalHeaderLabels(QStringList)
QTableWidget	QVariant getItem(int,int)
QTableWidget	void setItem(int,int,QVariant)
QTableWidget	int currentColumn() returns index of selected column
QTableWidget	int currentRow() returns index of selected row
QTableView	uiItem horizontalHeader()
QTableView	uiItem verticalHeader()

Please notice, that every method listed above is also valid for a widget, that is derived from the specific class


(derived in C++). Therefore the additional slots of *QWidget* hold for every other widget, since every widget is derived from *QWidget*.

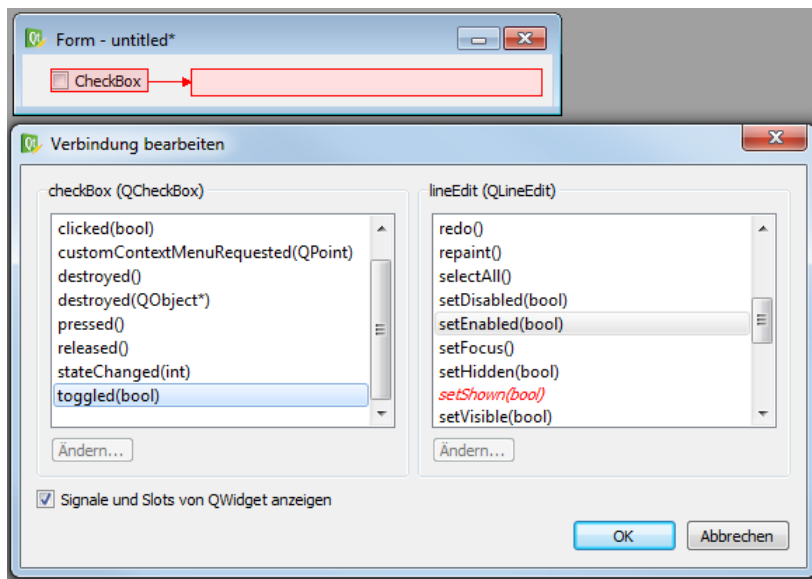
7.3.8 Connecting internal signals and slots in Qt Designer

If you want to connect the signal, emitted by any widget, with a slot from another widget, you will learn in this section how to do this. This type of connection can for instance be useful if you want to enable or disable certain widgets depending on the status of other ones, like the check-status of a checkbox. The following figure shows an user interface with a checkbox and a textfield. Let us define a signal-slot-connection, such that the textfield gets disabled if the checkbox is unchecked.



This type of gui-internal connections are completely done in **Qt Creator**. Therefore chose the “Signal and Slots”

editing mode, that is obtained by clicking the symbol  in the toolbar or by pressing *F4*. Then you can make a drag&drop connection between the emitting widget and the receiver-widget. After releasing the mouse button, the connection dialog, depicted in the following figure becomes visible:



Here you can choose which signal of the emitting widget should be connected with which slot of the destination. At the beginning, only slots and signals of the specific widget classes are visible. However, you can check the checkbox below, in order to also show the signals and slots of the inherited classes. Please make sure, that you only choose pairs of signals and slots which have the same parameter types. In our case, we connect the signal *toggled(bool)* with the slot *setEnabled(bool)*, which is the setter-method of the property *enabled*.

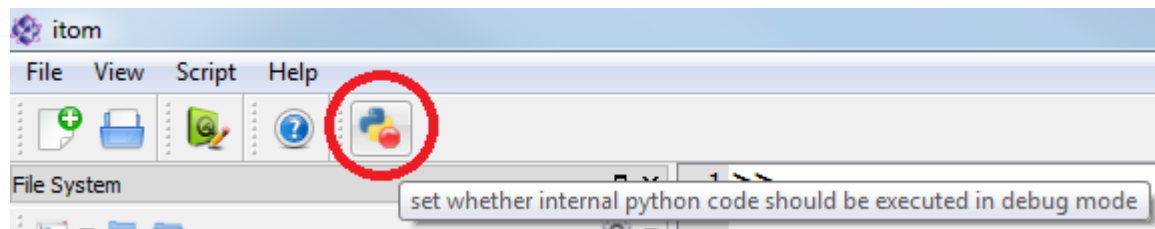
This example has also shown, that not only slots defined in the slot-section can be called as slots, but also every setter-method of any property can be called like every slot. However, in this case it is more convenient the property like described in section [Getting and setting properties](#).

7.3.9 Auto-connecting signals using python decorators

TODO

7.3.10 Debugging user interfaces and slot-methods

If you established a signal-slot-connection between an element of the GUI and a **Python**-method, you probably want to debug this method once the signal has been emitted. This is obtained by setting any breakpoint into the specific line and toggling the button *Run python code in debug mode* in the menu **Script** of **itom**.



7.3.11 Access the status bar of a main window

If one wants to access and modify the content of the status bar of a main window, the following steps need to be done:

Let's assume the main window is accessible via Python by the variable name **gui**, then

```
statusbar = gui.call("statusBar")
```

returns an instance of *uiItem* representing the status bar widget (Qt class *QStatusBar*). Check the Qt help to see that *QStatusBar* has the following slots (among others):

- **clearMessage()** clears the current text of the status bar
- **showMessage(const QString &message)** displays *message* in the status bar
- **showMessage(const QString &message, int timeout)** displays *message* in the status bar and hides it after the *timeout* given in milliseconds.

7.3.12 Hints and limitations

All methods described in this chapter explain how to create and use user-defined dialogs and windows using **Python** scripts in **itom**. Finally, all dialogs are created using the **Qt**-framework. The classes *itom.ui* and *itom.uiItem* finally are wrappers for the underlying **Qt**-system. Using pure python, similar things can also be obtained with the famous packages **PyQt** or **PySide**. However, in **itom** you must not use these packages. The reason is, that create a new main instance of the **Qt**-engine, that needs to be created in the main thread. This is the case, if **PyQt** or **PySide** is executed directly in **Python**. However using **itom Python** is embedded as scripting language, such that **itom** is executed in the main thread while **Python** is moved to its own additional thread. The reason is to enable the execution of longer scripts, while the main application **itom** still keeps reactive. Therefore, **Python** does not have access to the real main thread and it is forbidden to explicitly execute some GUI-related stuff in secondary threads. Therefore all methods in *itom.ui* and *itom.uiItem* have thread-safe implementations and communicate with an organization structure, that runs in the main thread of **itom**, in order to interact with all dialogs.

7.4 Custom Designer Widgets

Beside QtDesigner-Widgets for plots and figures (see *Plots and Figures*) some non-plotting widgets have been develop to give the user GUI-based access to itom specific objects and functions. The openSource-Widgets are:

- DataObjectTable
- dObMetaDataTable
- EvaluateGeometricsFigure

- MotorController

These widgets can be used like any other type of widget within an ui-dialog with 2 exceptions:

1. The ui-Dialog must be loaded and initilized within a itom-python context (e.g. script in itom).
2. Some properties are not accesable (DESIGNABLE) in the QtDesigner (e.g. actuator-handles) and must be set during initialization in itom-python.

To add such a widget to your ui-file, you can drag&drop them in the QtDesigner like any other widget.

7.4.1 DataObjectTable

The “DataObjectTable” can be used to visualize or edit a dataObject in a table based widget like in “Matlab”. The widget is not inherited from a AbstractDObject and can not be used for a live plot.

Properties

data: *ito::DataObject*, the dataObject to be shown

readOnly: *bool*, *DESIGNABLE*, enable write protection

min: *double*, *DESIGNABLE*, get/set minimum value

max: *double*, *DESIGNABLE*, get/set maximum value

decimals: *int*, *DESIGNABLE*, number of decimals to be shown within each cell

defaultCols: *int*, *DESIGNABLE*, number of column to be shown

defaultRows: *int*, *DESIGNABLE*, number of rows to be shown

horizontalLabels: *QStringList*, *DESIGNABLE*, list with labels for each column row

verticalLabels: *QStringList*, *DESIGNABLE*, list with labels for each shown row

7.4.2 dObMetaDataTable

The “dObMetaDataTable” can be used to visualize the metaData of a dataObject, e.g. to realize a measurement protocol . The widget is not inherited from a AbstractDObject and can not be used for a live plot.

Properties

data: *ito::DataObject*, the dataObject to be shown

readOnlyEnabled: *bool*, *DESIGNABLE*, enable write protection

detailedInfo: *bool*, *DESIGNABLE*, Toogle between basic and detailed metaData

previewEnabled: *bool*, *DESIGNABLE*, Add a small quadratic image downsampled from the dataObject as a preview to the meta data.

previewSize: *int*, *DESIGNABLE*, Set the preview size in pixels,

decimals: *int*, *DESIGNABLE*, number of decimals to be shown within each cell

colorBar: *QString*, *DESIGNABLE*, the name of the color bar for the preview, *not implemented yet*

7.4.3 EvaluateGeometricsFigure

The evaluateGeomtrics-widget is designed to load geometric definition stored in a float32 dataObject with a column-size of >10 elements and a row for each geometric element to display. Further more it allows the evaluation of geometric relations between the geometric primitives. It contains a tableView and although is inherited by AbstractDObject it should not be used for “liveVisualisation of dataObject”.

Properties

title: *QString, DESIGNABLE*, Title of the plot or ‘<auto>’ if the title of the data object should be used. *Not implemented yet* **valueUnit:** *QString, DESIGNABLE*, The value unit for the metrical calculations that is used within the plot. **titleFont:** *QFont, DESIGNABLE*, Font for title. *Not implemented yet* **labelFont:** *QFont, DESIGNABLE*, Font for labels. *Not implemented yet* **relations:** *ito::DataObject*, Get or set N geometric elements via N x 11 dataObject of type float32. **relationNames:** *QStringList, DESIGNABLE*, A string list with the names of possible relation. The first elements [N.A., radius, angle to, distance to, intersection with, length and area] are read only and are calculated with these widget. For external calculated values you can define custom names e.g. roughness.. **destinationFolder:** *QString, DESIGNABLE*, Set a default export directory. **lastAddedRelation:** *int, DESIGNABLE*, Get the index of the last added relation. **considerOnly2D:** *bool, DESIGNABLE*, If true, only the x & y coordinates are considered.

Slots

ito::RetVal addRelation(ito::DataObject importedData)

Add a relation to the current context. The relation must be

importedData, ito::DataObject: geometric element expressed by 1 x 4 dataObject of type float32.

ito::RetVal modifyRelation(const int idx, ito::DataObject relation)

Modify an existing relation addressed by the idx.

idx, int: Index of relation to modify

relation, ito::DataObject: geometric element expressed by 1 x 4 dataObject of type float32.

ito::RetVal addRelationName(const QString newName)

Add a new relation name to the relationNameList.

newName, QString: new relation name to be appended

ito::RetVal exportData(QString fileName, ito::uint8 exportFlag)

Export data to csv or xml

fileName, QString: Destination file name

exportFlag, int: Export flag, exportCSVTree = 0x00, exportCSVTable = 0x01, exportXMLTree = 0x02, exportCSVList = 0x03, showExportWindow = 0x10

ito::RetVal plotItemChanged(ito::int32 idx, ito::int32 flags, QVector<ito::float32> values)

Slot for direct connection between this widget and a plot (e.g. itom2dQwtPlot) to notify changes within the plotted geometry. Internal relations are automatically updated. External relation values (e.g. roughness) can not be updated automatically.

idx, int: Index of the modified geometric element

flags, int: Type (and meta properties) of geometric elements which was changed. If type differs from original type clear and refill is necessary.

values, QVector<ito::float32>: Geometric parameters of the modified geometric item.

ito::RetVal clearAll(void)

Clear all elements and relations in this plot.

Signals

None

7.4.4 MotorController

The “MotorController”-widget gives the user the some basic functions for generic motor positioning and position reporting within a complex GUI. The widget can be used for 1 to 6 axis and can be used `readOnly` or as a type of software control panel. The widget updated in a fixed interval (can be deactivated). During measurements the widget should be disabled to avoid user errors. A support for the 3DConnexion-Mouse is planed but *Not implemented yet*.

The motor should support up *slot: RequestStatusAndPosition* and *signal: actuatorStatusChanged* for semaphore free communication, but this is not necessary.

Properties

actuator: *ito::AddInActuator*, Handle to the actuator to be used, not `DESIGNABLE`

numberOfAxis: *int, DESIGNABLE*, Number of axis to be visible

unit: *QString, DESIGNABLE*, Base unit for spinboxes and movements, e.g. nm, micron, mm, m, km

readOnly: *bool, DESIGNABLE*, Toogle read only

autoUpdate: *bool, DESIGNABLE*, Toogle automatic motorposition update

smallStep: *double, DESIGNABLE*, Distances for the small step button, same value for plus and minus

bigStep: *double, DESIGNABLE*, Distances for the large step button, same value for plus and minus

absRel: *bool, DESIGNABLE*, Toogle between absolut or relative position display. Origin can be set via context menu.

allowJoyStick: *bool, DESIGNABLE*, Allow a software joystick, e.g. usb or gameport, not implemented yet.

Slots

void triggerActuatorStep(const int axisNo, const bool smallBig, const bool forward)

Trigger a step of axis *axisNo* with a distance either *bigStep* (*true*) or *smallStep* (*false*) and either *forward* (*true*) or *backwards* (*false*)

void actuatorStatusChanged(QVector<int> status, QVector<double> actPosition)

Internal slot for c++-Code connected to the corresponding signal of the actuator. Do not call this from python.

void triggerUpdatePosition(void)

Usually called by the internal timer, the context-menu or python to update the current motor position. Uses either Signal-/Slot-communication or invokes `getPos` blocking.

void guiChangedSmallStep(double value)

Internal slot if the small step size is changed within the GUI (spinbox)

void guiChangedLargeStep(double value)

Internal slot if the large step size is changed within the GUI (spinbox)

Signals

void RequestStatusAndPosition(bool sendActPosition, bool sendTargetPos)

Internal signal for c++-Code connected to the corresponding slot of the actuator.

7.5 Implement a more complex GUI in a plugin (C/C++)

This chapter will become a introduction how to programm widgets and qtDesigner widgets one day. But till then read the source code, copy an existing project or write an email to lyda@twip-os.com.

PLUGINS

This chapter contains information about the software and hardware plugin mechanism of **itom**. The software plugins may contain algorithms, written in C++, which can be called from any python script or another plugin. Furthermore, software plugins can also contain arbitrary user interfaces. This allows implementing complex dialogs and windows using all possibilities given by the Qt-framework. Hardware plugins allow to implement actuators, cameras, AD-converters and other devices. Then these devices are also accessible by any python script.

See the following sections in order to get more information about these plugins:

8.1 Basic concept

The basic **itom** -plugin concept

8.2 How to start and use a plugin

8.2.1 Hardware Plugins

In this section you will learn how to create one instance of any hardware plugin (actuator, dataIO...) by a python script or the GUI of **itom**.

Hardware plugins are mainly divided into the following main categories:

- **DataIO** contains all plugins that are related with any data input/output operation. Short, these are mainly
 - cameras and framegrabbers (organized in the subcategory **Grabber**),
 - AD or DA converters (subcategory **ADDA**) or
 - further input / output devices (subcategory **Raw IO**) like the serial port plugin
- **Actuator** contains all motor stages, piezo actuators or further actuator devices

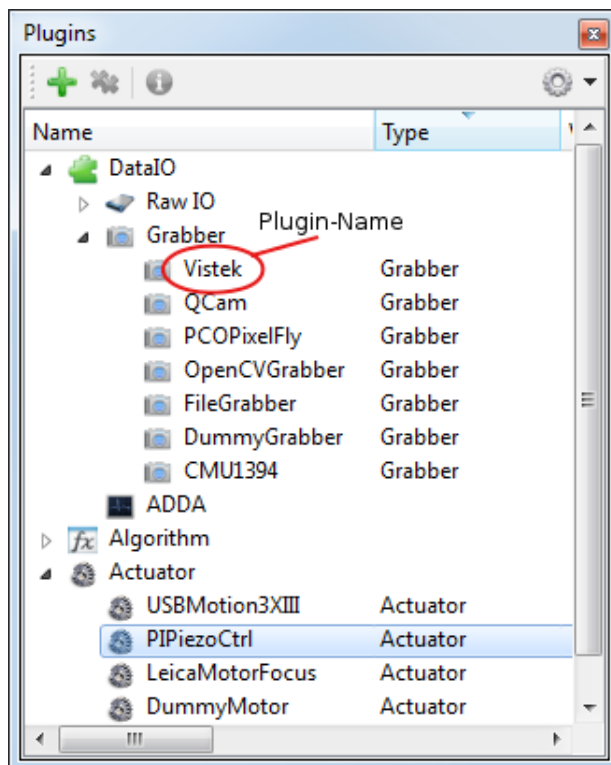
All valid plugins are listed in the plugin toolbox of **itom**

Script-based startup

In order to start a new instance of any hardware plugin using the python scripting language, search the plugin toolbox for the desired plugin and remember its plugin-name (case sensitive). Every plugin instance usually needs some mandatory and optional parameters for creating a new instance. Use the command `itom.pluginHelp()` in order to get information about a plugin including descriptions of its mandatory and optional parameters:

```
pluginHelp("pluginName")
```

The information is then printed out in the command line of **itom**.



Every hardware plugin is either represented by the class `itom.dataIO` or `itom.actuator`. By creating a new instance of one of these classes, a new instance of the corresponding plugin is created. The syntax of the constructor is:

```
myVar = dataIO("pluginName", mandatoryParameters, optionalParameters)
#or
myVar = actuator("pluginName", mandatoryParameters, optionalParameters)
```

where **mandatoryParameters** are all mandatory parameters (just place one value after the other one separated by commas) followed by the optional parameters. If you only want to indicate some of the optional parameters use the keyword-based argument passing of python:

```
serial = dataIO("SerialIO", 1, 9600, "\n", parity = 1)
# opens a serial port device at COM1 with 9600 bauds,
# an endline character '\n' and an odd parity as single optional parameters
```

Once an instance of a plugin is created, the corresponding entry in the plugin toolbox obtains a new child item. In **Python**, the variable you used for creating the plugin, is created and can now be used for controlling the plugin. As long as any variable(s) in **Python** still hold a reference to this plugin, its background color in the plugin toolbox is yellow.

In order to close/delete an instance of a plugin you need to delete all variables in **Python** that are referencing this plugin, using the command `del ()`:

```
del serial
```

Note: Since **Python** has a garbage collection mechanism it is not assured that any object is immediately deleted once you delete the corresponding variable(s) in the **Python** workspace. It is really done if the garbage collection process is started which happens in periodical steps. You can force it using the method `gc.collect()` from the builtin module `gc`.

GUI-based startup

TODO

If you want to program your own plugin, see the following sections of the documentation:

8.3 Development under C++

The functionality of *itom* can be enlarged by programming plugins (sometimes also called add-ins). Such a plugin is written in C++ and can be integrated into *itom* using the possibilities which are given by *Qt* (plugin management of *Qt*). Additionally each plugin must fit to one of three possible interface classes, such that *itom* is able to communicate with this plugin. Finally, the resulting plugin is compiled as a *dll*-file (on Windows) or as an appropriate *a*-file on Linux-based machines and must be located in the *plugin*-folder of *itom* or any subfolder. Then, the plugin is automatically recognized at startup of *itom*.

For programming plugins, you can and should use basic structures, offered by **itom**:

8.3.1 RetVal - The return type of itom methods

The class **RetVal** is used for creating, transmitting and handling return values of methods in **itom** and plugins.

For using this class, include the file *retVal.h* from the folder *include/common* of the **itom** SDK directory

```
#include "retVal.h"
```

and link against the library **itomCommonLib**, that is also contained in the SDK.

Any return value consists of the following main components:

- **error state (enumeration ito::tRetVal)**
 - retOk
 - retWarning
 - retError
- error number (user-defined error number; this number has no further functionality yet)
- error message (NULL or the warning or error message of the return value)

You can create a variable of type **RetVal** using different constructors or the static method **RetVal::format**.

ito::RetVal::RetVal (tRetVal *retValue* = retOk)
creates a return value with the default state *retOk*, if not otherwise stated. (constructor)

ito::RetVal::RetVal (int *retValue*)
creates a return value with the given state. (constructor)

ito::RetVal::RetVal (ito::tRetVal *retValue*, int *retCode*, const char **pRetMessage*)
creates a return value with a given state, error number and error message. If you don't want to indicate an error message, set that value to NULL. (constructor)

static ito::RetVal ito::RetVal::format (ito::tRetVal *retValue*, int *retCode*, const char **pRetMessage*, ...)

Use this static method to create a new return value where the message string can contain placeholders, known by the ordinary methods **sprintf**, ... The values filled into these placeholders are then appended as additional parameters to this method.

RetVal &operator= (const RetVal *rhs*)

By this operator you can assign a new return value to this return value.

Additionally you can use the mathematical operators **+** and **+=** to add a return value to an existing instance of class **RetVal**. The resulting return value keeps unchanged if the state of the added return value is less critical than the internal state, e.g. a state *retOk* is less critical than *retError* as well as *retWarning* is less critical than *retWarning*. If both states are the same, the return value is unchanged, too. In any other cases the state, error number and error message of the added return value is used for the new return value.

By this mechanism you can create chains of return values in the following form:

```
ito::RetVal retValue = ito::retOk
retValue += method1()
retValue += method2()
retValue += method3()

if(retValue.containsError())
{
    print("error while executing some methods");
}
```

The actual status of the return value can be obtained using the following methods:

int **containsWarning** ()

Returns true if the error state is equal to retWarning, that means the worst error state which has been added to this return value has been retWarning.

int **containsError** ()

Returns true if the error state is equal to retError, that means the worst error state which has been added to this return value has been retError.

int **containsWarningOrError** ()

Returns true if the worst error state, which has been added to this return value has been unequal to retOk.

char ***errorMessage** ()

Returns a zero-terminated string containing the actual error-message of this return value or a zero-terminated, empty string if no message has been set (Caution: in **itom** <= 1.1.0 this method returned NULL in the latter case).

Additionally you can use the comparison operators == or != to compare the error state of two return values or the error state of one return value with a given error state.

Note: For a full reference of the class **RetVal** see [RetVal - Reference](#).

RetVal - Reference

class **ito::RetVal**

Class for error value management.

The *RetVal* class is used for handling return codes. All classes should use this class. In case an error occurs, only the first error is stored and will not be overridden by potentially subsequent occurring errors.

Public Functions

RetVal (ito::tRetValue *retValue*, int *retCode*, const char * *pRetMessage*)

constructor with retValue, retCode and errorMessage

Parameters

- *retValue* - type of *RetVal*; for possible values see tRetValue
- *retCode* - user definable return code
- *pRetMessage* - error message to be passed or NULL, string is copied Makes a deep copy of *RetVal*, i.e. a copy of the error message

RetVal & **operator=** (const *RetVal* & *rhs*)

assignment operator, copies values of rhs to current *RetVal*. Before copying current errorMessage is freed

RetVal & **operator+=** (const *RetVal* & *rhs*)

Concatenation of *RetVal* “Adds” RetVals, i.e. returns the most serious error. In case of equally serious errors the first is retained

RetVal **operator+** (const *RetVal* & *rhs*)

Concatenation of *RetVal* See operator *RetVal::operator+=*

char **operator==** (const *RetVal* & *rhs*)

equality operator compares retValue with with retValue of rhs *RetVal*. For possible constant values see tRetVal

char **operator!=** (const *RetVal* & *rhs*)

unequality operator compares retValue with with retValue of rhs *RetVal*. For possible constant values see tRetVal

char **operator==** (const tRetVal *rhs*)

equality operator compares retValue with tRetVal constant. For possible constant values see tRetVal

char **operator!=** (const tRetVal *rhs*)

unequality operator compares retValue with tRetVal constant. For possible constant values see tRetVal

int **containsWarning** () const

checks if any warning has occurred in this return value (true), else (false)

int **containsError** () const

checks if any error has occurred in this return value (true), else (false)

int **containsWarningOrError** () const

checks if any warning or error has occurred in this return value (true), else (false)

const char * **errorMessage** () const

returns zero-terminated error message or empty, zero-terminated string if no error message has been set

8.3.2 ItomSharedSemaphore

In **itom** different main components are executed in different threads.

Note: What is a thread

In a modern operating system, different programs can run simultaneously. Then, every program is executed in its own process and the operating system is responsible such that each process continuously gets some processing-time on the processor. If your computer has multiple processors, than the operating system can assign processing time of different processes to different processing units.

Every program itself can now run different components of its own software in different threads. Thread are comparable to processes on operating system level, however threads can share memory, such that every thread can have access to a common set of global variables, instances... The advantage is, that a thread can execute a time-consuming calculation while the other thread is responsible for the graphical user interfaces, which now is not blocked while the calculation thread is working.

For more information about threads see [Wikipedia \(english\)](#) or [Wikipedia \(german\)](#)

In **itom**, the following components run into a different thread:

1. Main programm including all graphical user interface components (this is a restricting of windows operating systems). This thread is called “main thread”.
2. Python scripting engine. All python scripts are executed in a second thread. Therefore you should not directly use any 3rd-party python modules which open any graphical user interface (like *PyQt*, *tkInter* ...).
3. Every *DataIO* or *Actuator*-plugin is executed in its own thread. Every algorithm is executed in the thread of the calling instance (e.g. main thread or python thread). All widgets provided by any *Algo*-plugin are always executed in the *main thread* (see restricting of point 1).

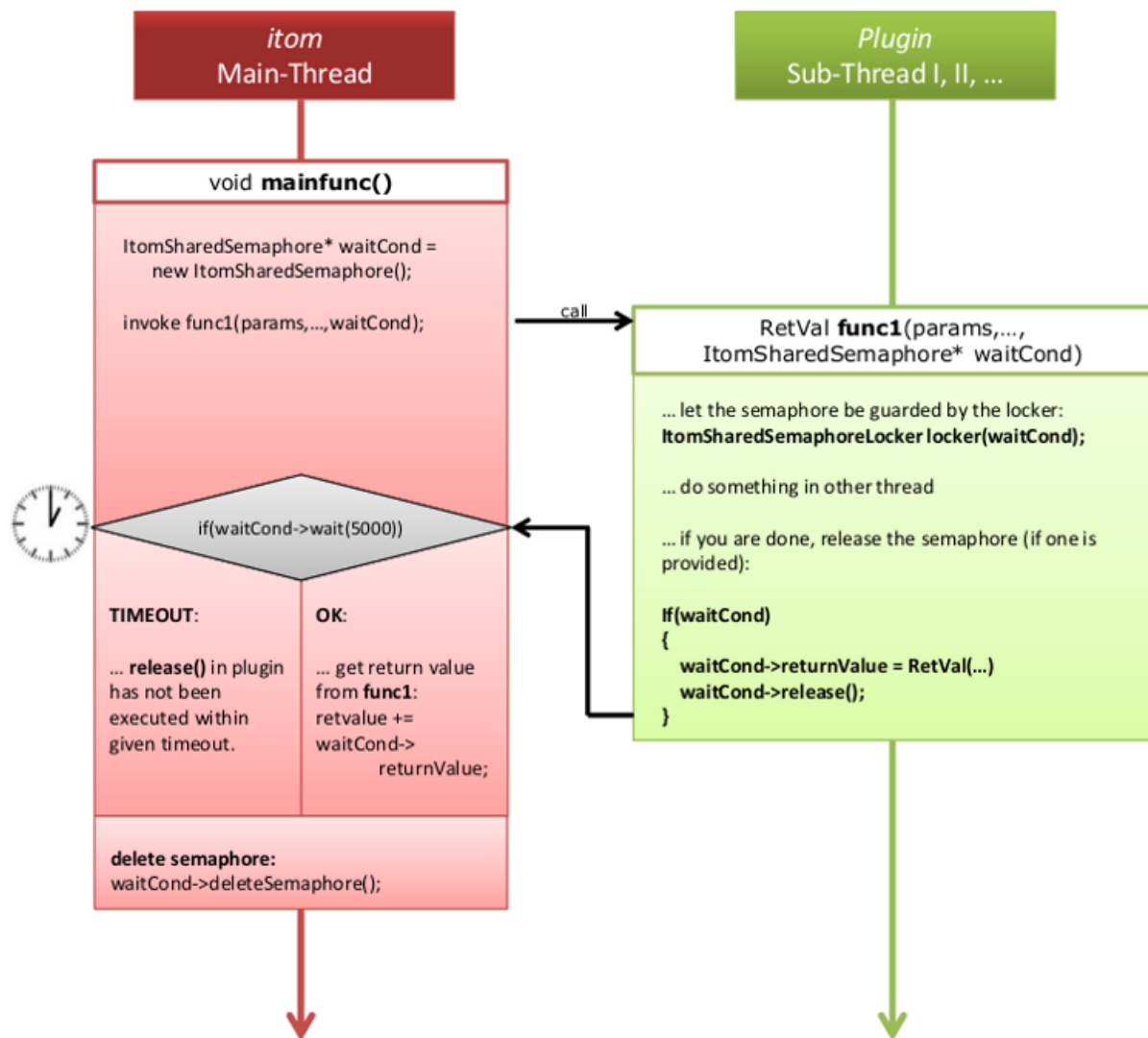


Fig. 8.1: Figure: Scheme of a communication between **itom** (main thread) and any plugin-method (plugin is running in another thread)

In figure *Figure: Scheme of a communication between itom (main thread) and any plugin-method (plugin is running in another thread)*, a common communication between a method **mainfunc** which is executed in **itom**'s main thread (e.g. in *AddInManager* and another method **func1** of a plugin is depicted. The plugin is of type *actuator* or *dataIO* and therefore runs in its own thread. **mainfunc** is calling the method **func1**. Usually the calling method has to wait until the called method (**func1**) has been finished (or until the most important parts of **func1** have been executed). In order to consider cases where the plugin-method is not answering within a certain timeout, **itom** has integrated a mechanism such that the waiting-process in the main-function can be stopped after that the timeout time expired. This mechanism is realized by the class **ItomSharedSemaphore**, which is defined in the file *sharedStructuresQt.h* (folder *common*).

ItomSharedSemaphore consists of the following elements:

QSemaphore *m_pSemaphore

ItomSharedSemaphore internally contains an instance of *QSemaphore*, the platform independent semaphore provided by **Qt**. A semaphore can be considered as set of *gaming piece*. Any method executed within different threads can take one or more *gaming pieces* from the common semaphore. If there are not enough *gaming piece* in the storage, the method has to wait until the necessary number of *gaming pieces* become available. Any method which exits some "critical sections" in the code release their previously picked *gaming pieces*, such that other instances can repick them. In most cases in the communication between **itom** and a plugin-method, the semaphore contains one *gaming piece*, since only two partners are participating in the communication: the method in the main thread and the called method in the plugin-thread. The method in the plugin-thread is picking the *gaming piece* and the calling method has to wait until the *gaming piece* gets back to the storage of the semaphore. This is achieved by *releasing* the *gaming piece* in the called method.

int m_numOfListeners

This is the number of participants at a communication process (without the calling method). Usually only one method is called, therefore this value is usually equal to 1. This value usually is automatically set.

int m_instCounter

This value indicates how many *gaming pieces* are out of the storage. If the called method is still executing, this value is equal to one. If the called method already release the semaphore, hence *gaming piece*, the value drops to zero.

bool isCallerStillWaiting

This method returns true if the calling method (*here*: the method in main thread) is still waiting that the called method has released the semaphore, else the method returns false.

ito::RetVal returnValue

The called method in the plugin-thread can set that globally accessible return value and the calling method can read its value. That return value is the single way to return the success or error message to the calling method, since no return value of the called method (slot) can directly be returned.

Scheme of the inter-thread communication

1. The calling method **mainfunc** creates an instance of *ItomSharedSemaphore*:

```
ItomSharedSemaphore *waitCond = new ItomSharedSemaphore();
```

2. Method **func1** is called in the plugin-thread. This can only be done if **func1** is declared as **slot** (*signal-slot-system* of **Qt**). The call is executed using the method **invokeMethod** of *QMetaObject*:

```
QMetaObject::invokeMethod(instance-of-plugin, "func1", Q_ARG(paramType1, paramValue1), ..., Q_ARG(paramTypeN, paramValueN));
```

For the use of the macros **Q_ARG** and the method **invokeMethod**, see also the documentation of **Qt**.

3. Then **func1** is executing and their last parameter *waitCond* contains the pointer to the instance of *ItomSharedSemaphore*.
4. If **func1** has been finished, you should write the return value to the instance of *waitCond* and release the semaphore, hence, return the *gaming piece* to the global storage of the semaphore:

```

    if(waitCond) //can also be NULL, therefore check it
    {
        waitCond->returnValue = ito::RetVal(ito::retOk)
        waitCond->release()
    }

```

5. The caller has to wait until the semaphore has been released or a timeout expired. This is done by the member **wait** of **ItomSharedSemaphore**. The single argument of that method is the timeout in milliseconds. Usually you can use the global variable **PLUGINWAIT**, which is set to **5 seconds**. The *wait*-method returns *true* if the semaphore has been released within the timeout, else it returns false:

```

    if(!waitCond->wait(PLUGINWAIT))
    {
        //timeout occurred
    }

```

5. Finally all participants at the communication process (*here*: caller and called method) have to delete the semaphore. Be careful: This can not be done by simply **deleting the pointer to waitCond**. Instead both the caller and the called method have to execute the following command, hence, call the method **deleteSemaphore** of the semaphore pointer:

```

waitCond->deleteSemaphore()

```

6. After that the last participant at the communication process deleted the semaphore, it is really deleted by **itom**. Then you don't have access to the semaphore any more, hence, you also don't have access to the internal return value.

In order to simplify the process of deleting the semaphore, both the caller and the calling method can also create a variable of type **ItomSharedSemaphoreLocker**, where the pointer to **waitCond** has to be given as constructor-argument. If this variable finally is destroyed, which is automatically done if the method is finished - even after that the *return* command has been executed - it calls the *deleteSemaphore*-method of *ItomSharedSemaphore* with *waitCond* as argument.

Then the scheme of the caller is:

```

ItomSharedSemaphoreLocker locker(new ItomSharedSemaphore())
QMetaObject::invokeMethod(plugin-instance, "func1", Q_ARG(paramType1,paramValue1), ..., Q_ARG(ItomSharedSemaphoreLocker,locker))

if(!locker.getSemaphore()->wait(PLUGINWAIT))
{
    //timeout
    retvalue += ito::RetVal(ito::retError,0,"timeout while executing method");
}
else
{
    retvalue += locker.getSemaphore()->returnValue;
}

```

and the scheme of the called method **func1** is:

```

void func1(type1 param1, ..., ItomSharedSemaphore* waitCond)
{
    ItomSharedSemaphoreLocker locker(waitCond)
    ... do some calculation ...

    if(waitCond)
    {
        waitCond->returnValue = ito::retOk;
        waitCond->release();
    }

    return
}

```

Note: For more information about the implementation of class **ItomSharedSemaphore** see [ItomSharedSemaphore - Reference](#) or the doxygen documentation for **itom**.

ItomSharedSemaphore - Reference

class **ItomSharedSemaphore**

semaphore which can be used for asynchronous thread communication. By using this class it is possible to realize timeouts.

This semaphore is usually applied if any method invokes another method in another thread and should wait for the called method being terminated or the waiting routine drops into a possible timeout. Therefore, the calling method must create an instance of [ItomSharedSemaphore](#) with a number of listeners equal to one. Then, the pointer to [ItomSharedSemaphore](#) is transmitted to the called method (usually as last argument). If the called method is done or wants the caller to continue the release-method of [ItomSharedSemaphore](#) is called. The calling method calls the wait-method of the semaphore which blocks the method until the semaphore is released. Finally, both the caller and the calling method must call [ItomSharedSemaphore::deleteSemaphore](#) in order to decrease the reference counter from two to zero, which allows the system to delete the semaphore.

Consider to guard an instance of [ItomSharedSemaphore](#) by the capsule-class ItomSharedSemaphoreLocker both in the caller and calling method, such that the decrease of the reference counter is executed if the ItomSharedSemaphoreLocker-variables are deleted, e.g. if they run out of scope, which is even after the return-command of any method.

Public Functions

ItomSharedSemaphore (int *numberOfListeners* = 1)
 constructor

A new [ItomSharedSemaphore](#) is created and the underlying semaphore is already locked with a certain number. This number depends on the number of listeners (usually: 1). A listener is a method which is called in another thread. The caller creates the [ItomSharedSemaphore](#) in order to wait until the listener decreases the lock level of the semaphore by one or a certain timeout time has been reached.

Parameters

- *numberOfListeners* - (default: 1) are the number of different methods in other threads which should release this semaphore before the caller can go on.

~ItomSharedSemaphore ()
 destructor (do not call directly, instead free the semaphore by [ItomSharedSemaphore::deleteSemaphore](#))

bool **wait** (int *timeout*)
 The call of this method returns if a certain timeout has been expired or every listener released the semaphore.

bool **waitAndProcessEvent** (int *timeout*, QEventLoop::ProcessEventFlags *flags* = QEventLoop::AllEvents)
 mutex initialization
 The call of this method returns if a certain timeout has been expired or every listener released the semaphore.

void **release** ()
 decreases the number of locks by one

int **available** () const
 checks whether the semaphore is still locked or not

bool **isCallerStillWaiting** ()

indicates whether caller-method is still waiting that the lock is released by the listener(s).

void **deleteSemaphore** (void)

static method to decrease the reference counter of any *ItomSharedSemaphore* or delete it if the reference counter drops to zero

Public Members

ito::RetVal **returnValue**

public returnValue member variable of *ItomSharedSemaphore*. This return value can be used to return the result of any called method (listener) to the caller. Please access this value in caller only if the wait-method returned with true. Write to returnValue in called method (listener) before releasing the semaphore. This is important, since the returnValue is not fully thread safe.

8.3.3 DataObject

The class **DataObject** (part of the library **dataObject**) provides a n -dimensional matrix that is used both in the core of **itom** as well as in any plugin. The n -dimensional matrix can have different element types. These types and their often used enumeration value are defined in the file *typeDefs.h* and are as follows:

Typedef	Enumeration	Description
ito::int8	ito::tInt8	8bit, signed, fixed point
ito::uint8	ito::tUInt8	8bit, unsigned, fixed point
ito::int16	ito::tInt16	16bit, signed, fixed point
ito::uint16	ito::tUInt16	16bit, unsigned, fixed point
ito::int32	ito::tInt32	32bit, signed, fixed point
ito::uint32	ito::tUInt32	32bit, unsigned, fixed point
ito::float32	ito::tFloat32	32bit, single-precision floating point
ito::float64	ito::tFloat64	64bit, double-precision floating point
ito::complex64	ito::tComplex64	real and imaginary part is float32 each
ito::complex128	ito::tComplex128	real and imaginary part is float64 each

The last two dimensions of each DataObject are denoted *plane* and physically correspond to images. Since no one-dimensional DataObject is available, each DataObject at least consists of one plane. In order to also handle huge matrices in memory, usually the different planes are stored at different locations in memory. Internally, each plane is an OpenCV matrix of type **cv::Mat_<type>**, derived from **cv::Mat**. Therefore every plane can be used with every operator given by the **OpenCV**-framework (version 2.3.1 or higher). This kind of DataObject and its way of allocating memory is called *non-continuous*.

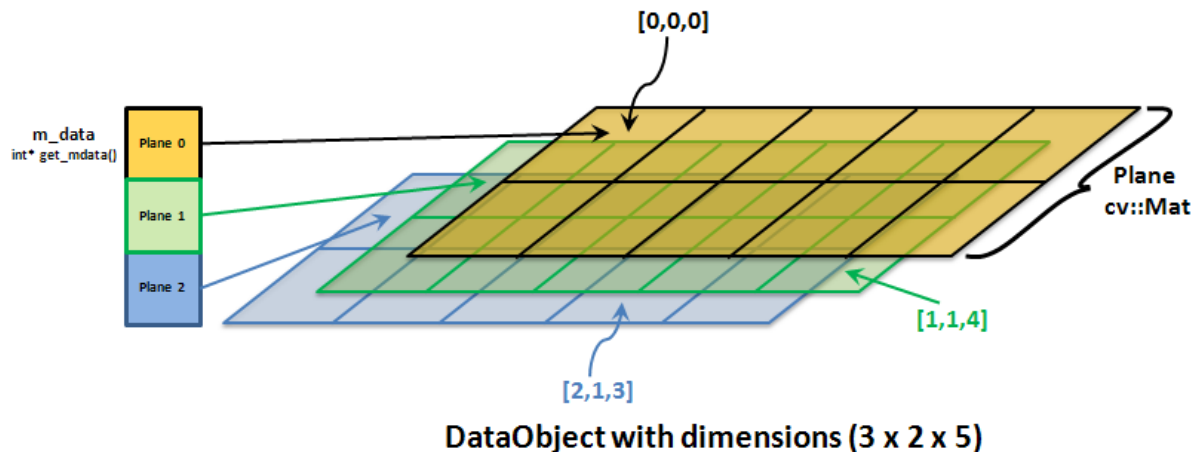
In order to make the *DataObject* compatible to matrices that are allocated in one huge memory block (like Numpy arrays), it is also possible to make any *DataObject* continuous. Then, a huge data block is allocated, such that all planes lie consecutively in memory. This reallocation is implicitly done, when creating a Numpy-array from a non-continuous DataObject.

DataObject can be declared in different possible ways with different dimensions and different data types.

Let's take an example of a 3x2x5 data object. It can be imagined as shown in the figure below.

As we can see in this figure, each plane is of a type **cv::Mat** class from **OpenCV** library which we know. The internal index of a specific plane can safely be retrieved using the method **seekMat()**. Usually the i -th plane has got the index i , however in case of data objects representing a subpart or region of interest of another data object, the i -th plane with respect to the current region of interest can in reality have a bigger index than i . The pointers to all planes are stored in one linear vector, represented by the array member **m_data**. It is accessible via **get_mdata()** and is of type **int***. However, it can directly and safely be type-casted to **cv::Mat*** or **cv::Mat_<Type>**. Please read the section *Direct Access to the underlying cv::Mat* to understand this concept in detail with a working example.

The following code creates an empty data object with no dimensions (0) and no type (0).



```

1 ito::DataObject d0;
2 std::cout << "empty data object: \n";
3 std::cout << " dimensions: " << d0.getDims() << "\n";
4 std::cout << " type: " << d0.getType() << "\n" << std::endl;

```

The number of dimensions of this data object are obtained by **getDims()**, whereas its type value in terms of the enumeration above is returned via **getType()**.

The following code creates a 2 dimensional data object of dimensions Y=2, X=5 and of type *float32*.

```

1 ito::DataObject d1(2,5, ito::tFloat32);
2 std::cout << "2x5 data object, float32: \n";
3 std::cout << " dimensions: " << d1.getDims() << "\n";
4 std::cout << " type: " << d1.getType() << "\n";
5 std::cout << " size: " << d1.getSize(0) << " x " << d1.getSize(1) << "\n";
6 std::cout << " total: " << d1.getTotal() << "\n";
7 std::cout << d1 << std::endl;

```

The size of the specific dimensions is obtained by **getSize** where the argument is the index of the dimensions (*x* is always the last dimension with the biggest index value). **getTotal** returns the total number of elements within this data object.

Creating a data object

For creating a data object in C++, there are different constructors available. They are discussed in this section.

An empty data object is created using the argument-less, default constructor:

```
ito::DataObject d1;
```

It has no dimensions and contains no elements. For creating a two or three dimensional data object of a desired type, filled with arbitrary, but not random values, use one of the following constructors:

```
ito::DataObject(const int sizeY, const int sizeX, const int type); //type is one of the enumerati
ito::DataObject(const int sizeZ, const int sizeY, const int sizeX, const int type, const unsigned
```

The optional *continuous* parameter indicates, whether a continuous object (one block in memory) should be allocated (1), or not (0, default). For higher dimensional objects, there is a constructors that requires an allocated integer array that contains the sizes of all dimensions:

```
ito::DataObject(const unsigned char dimensions, const int *sizes, const int type, const unsigned
//e.g.
```



```
int sizes[] = {3,4,5,2};
ito::DataObject d1(4,sizes,ito::tFloat32);
```

One other important constructor is - of course - the copy constructor. It creates a shallow copy of an existing data object. A shallow copy means, that both data objects share the same data (hence the array itself), but the header information is separated (dimensions, sizes...). All meta information is shared as well (axes descriptions, scales, protocol...). If a value of the one object is changed, the corresponding value in the other object is changed as well. The principle of using shallow copies is a common principle in C programming and highly used in OpenCV. It speeds up the calculation and requires less memory. On the other hand, you need to take care about the values. If you want to have a so called deep copy of an object, use the **copy** method.

If you know what you do, you can also create a data object from multiple *cv::Mat* of the same type and size. If so, you need to have an array of all *cv::Mat* (each corresponding to one plane). The created data object then creates a fast shallow copy of all planes and uses them:

```
ito::DataObject(const unsigned char dimensions, const int *sizes, const int type, const cv::Mat* planes)
//e.g.
cv::Mat planes[] = { cv::Mat::ones(50,50,CV_8U), cv::Mat::zeros(50,50,CV_8U) };
int sizes[] = {50,50};
ito::DataObject d1(2, sizes, ito::tUInt8, planes, 2);
```

Furthermore, there are several constructors to create data objects whose values are already set to the values of a given array. The reader is referred to the detailed documentation of class **ito::DataObject** for this.

Addressing the elements of a data object

Now, we know how to create a data object, so lets have a look at how can one address the elements of a data object. Sometimes it is necessary to read or set single values in one matrix, sometimes one want to access all elements in the matrix or a certain subregion. Therefore, the addressing can be done in one of the following ways:

Direct access of one single element of a Data Object using `at<_Tp>()` method

```
1 ito::DataObject d1(2,5, ito::tFloat32);
2 d1.at<ito::float32>(0,1) = 5.2;
3 std::cout << "d1(0,1) = " << d1.at<ito::float32>(0,1) << "\n" << std::endl;
```

Here, the addressing is done by the member method **at()**, which is pretty similar to the same method of the **OpenCV** class **cv::Mat**. The **at()** method can either be used to get the value at a certain position or to set value at that position in a data object. There are special implementations of **at()** for addressing values in a two- or three-dimensional data object, where the first argument always is the **z-index** (3D), followed by the **y-index** and the **x-index**. All indices are zero-based, hence the first element can be referred by addressing **0th position** in every dimension.

Note: The **at()** method is templated where the template parameter must correspond to the type of the corresponding data object.

Let's try to summarize some pros and cons of this method.

Advantages

- This method gives flexibility to a developer to directly access any element of a data object.
- A developer can also access a part of a data object as well using **at()** method as described in [Direct Access to the underlying cv::Mat](#).

Drawbacks

- Developer has to implement the code under the nest of **if...else** conditions if one needs to access the whole data object.

- It is slow for accessing a lot of values of the matrix compared to the other possible methods.

Addressing elements of a data object using row pointer

When one needs to iterate through certain regions of a data object, then the previous method of accessing a data object using **at** method seems quite insufficient. In such case, one can define a row pointer for each row in matrix and work with row pointer to address elements of a data object in the following way.

```

1 ito::DataObject d1(3,5, ito::tInt16);
2 int planeID = d1.seekMat(0);    //get internal plane number for the first plane
3 ito::int16 *rowPtr = NULL;
4 int height = d1.getSize(0);
5 int width = d1.getSize(1);
6 for(int m = 0; m < height; m++)
7 {
8     rowPtr = (ito::int16*)d1.rowPtr(planeID,m);
9     std::cout << "Row " << m << ":";
10    for(int n=0; n < width; n++)
11    {
12        rowPtr[n] = m;           //accessing each element of data object with row pointer
13    }
14 }
15 std::cout << d1 << std::endl;

```

Here, **seekMat()** method gets the internal plane number of the 1st plane in line #2. In line #8, the pointer to the data array of the m-th row in the 2D-plane is obtained and saved in *rowPtr*. By iterating through the array given by the *rowPtr*, each element in this row can be read and set to a specific value.

To use this row pointer method for data objects more than 2 dimensions, following code can be used.

```

1 ito::int16 *rowPtr1= NULL;
2 int dim1 = d1.getSize(0);
3 int dim2 = d1.getSize(1);
4 int dim3 = d1.getSize(2);
5 int dim4 = d1.getSize(3);
6 int dim5 = d1.getSize(4);
7 int dataIdx = 0;
8 for(int i=0; i<dim1; i++)
9 {
10    for(int j=0; j<dim2; j++)
11    {
12        for(int k=0; k<dim3; k++)
13        {
14            dataIdx = d1.seekMat(i*(dim2*dim3) + j*dim3 + k);
15            for(int l=0; l<dim4; l++)
16            {
17                rowPtr1= (ito::int16*)d1.rowPtr(dataIdx,l);
18                for(int m=0; m<dim5; m++)
19                {
20                    //Assigning unique value to each element of d1.
21                    rowPtr1[m] = yourValue;
22                }
23            }
24        }
25    }
26 }

```

Note: Here **dataIdx** represents the number of the plane in the matrix. The formula in line #14 assigns a non repeating increasing value to **dataIdx** such that each plane of the data object can be pointed out without any overlapping.

Note: Usually it is not allowed to use the *rowPtr* to access the items of one of the following rows, since it is not sure if the elements of the next row directly follow to the last value of this row. For instance, this is not the case, if the current plane represents a region of interest of a bigger plane. You can only iterate through one entire plane using the row-pointer of the first row if the plane is continuous. This can be checked by the continuous property of the *cv::Mat* that represents this plane.

Some advantages and disadvantages of using this method are given in the section below.

Advantages

- This method is the most efficient way to access the data object.
- This method gives flexibility to access some rows or the full data object at once.

Drawbacks

- Complex implementation. One needs deep understanding of pointers to implement this method to access data object.
- This is not an advisable method if one needs to access a few elements of the data object which are not in sequence.

Assigning a single value to all elements of a data object

One can assign a single value to all elements of a data object using the assignment operator “=” in the following way. Here we will also have a look on how to declare a 5 dimensional data object and assign a single floating point value to each element of the data object.

```
1 int temp_size[] = {10, 12, 16, 18, 10};
2 ito::DataObject d1(5,temp_size,ito::tFloat32);
3 d1 = 3.7;
4 std::cout << d1 << std::endl;
```

Direct access to the underlying cv::Mat

In some cases, one needs to assign values of elements of a data objects based on some portion of another data object. This can be done by using this method of accessing the underlying matrix (*cv::Mat*) of a data object directly. Following example shows the method to access underlying planes in multidimensional matrices.

```
1 // 4 x 5 x 3 DataObject, int16
2 ito::DataObject d4(4,5,3,ito::tInt16);
3 std::cout << "DataObject (4x5x3), int16 \n" << std::endl;
4 d4 = 3; //assign value 3 to all elements
5 //access to the third plane (index 2)
6 int planeID = d4.seekMat(2);
7 cv::Mat *plane3 = (cv::Mat*)d4.get_mdata()[planeID];
8 std::cout << "OpenCV plane" << std::endl;
9 std::cout << *plane3 << std::endl;
10 //accessing second line in plane3
11 ito::int16* rowPtr2 = (ito::int16*)plane3->ptr(1);
12 //regions of interest
13 //d5 = d4[1:3,0:2,:];
14 ito::Range ranges[3] = { ito::Range(1,3), ito::Range(0,2), ito::Range::all() };
15 ito::DataObject d5 = d4.at(ranges);
16 d5 = 7;
```

Let's try to analyse the code above. As we can see in line #6, we used **seekMat()** method to retrieve the plane id of 3rd plane in 3 dimensional matrix d4.

line #7 declares a pointer variable *plane3* of type *cv::Mat* to hold the contents of plane 3 of data object d4. Line #11 declares a row pointer to point a particular row in plane 3 of data object d4 as a revision to the previous method of accessing elements of a data object using row pointer.

line #14 defines the exemplary ranges to create a new data object d5 from a part of data object d4, which is done in line #15 with the use of `at()` method.

The other way to perform the same operation of line #14 is shown below.

```
1 ito::Range *ranges = new ito::Range[3];
2 ranges[0] = ito::Range(1,3);
3 ranges[1] = ito::Range(0,2);
4 ranges[2] = ito::Range::all();
5 delete ranges;
```

This code shows the way to modify ranges individually, which can be very useful if one needs to modify this range later in this code to work on other data objects perhaps.

Note: Please consider, that the first index of the range is the first zero-based index inside of the selected range. The second value is always **one** index after the last index inside of the region of interest. This is very important!! This behaviour is somehow unintuitive, however similar to OpenCV and Python.

Note: `get_mdata()` is a function declared under *DataObject* class. It returns pointer to vector of *cv::_Matrices*.

Accessing all elements of a data object using iterators

There are two classes defined, called **DObjIterator** and **DObjConstIterator** respectively, under the namespace **ITOM**, which support the developer with an easy way to iterate through the whole data object. This method can be used only if one needs to iterate through all elements of a data object at once. Following code snippet shows the example of this method.

```
1 int temp = 0;           // Temporary variable for indexing some arrays used in this test.
2 ito::DataObject d6(21,13,ito::tInt16); // Declaring a 21 x 13 data object with data type int16.
3 ito::DObjIterator it;   // Declaration of DObjIterator
4 for(it=d6.begin();it!=d6.end();++it)
5 {
6     *((ito::int16*)(*it_2d)) = cv::saturate_cast<ito::int16>(temp++); // Assigning a unique v
7 }
```

As can be seen in the code above, line #2 declares a 21x31 data object d6 of type int16. Line #3 declares an iterator object **it** of class **DObjIterator**. *DataObject* class contains **begin()** and **end()** methods to work with iterators. A brief description to this methods can be found under *DataObject - Reference* document. These methods contains pointers to the first and last elements of any data objects respectively. Line #4 makes a meaningful use of these methods in for loop to iterate through the data object **d6**. We first initiate the iterator **it** with the pointer returned by **d6.begin()**, iterate through the whole data object increasing the iterator value by one in each iteration till the pointer value in iterator **it** reaches the pointer value of the last element of the data object checking the condition **it!=d6.end()**.

Advantages

- This method is a compromise between its usability with ease and performance on execution level. Integration of this method in code is fast and easy.
- Developer does not think about **if...else** conditions to decide the boundaries of Region of Interest to access any data object.

Drawbacks

- Performance degrades against the method *Addressing elements of a data object using row pointer*.
- It is not advisable to use this method if one needs to access some part or a single element of a data object.

Working with data objects

Now, let's have a look on various methods to work with data objects.

Creating Eye Matrix

Any square data object might need to be converted in eye matrix during many operations in matrix calculations. This can be quickly done using function **eye()** declared under *DataObject* class. Syntax for the function **eye()** is shown below.

dataObjectName.eye(noOfDimensions, dataType);

Return Type: void

To understand the use of this function, following is an exemplary code given. Let's have a look at it.

```
1 ito::DataObject *d2 = new ito::DataObject();
2 d2->eye(3, ito::tInt8);
3 std::cout << "3x3-eye matrix (int8)" << *d2 << std::endl;
4 delete d2;
5 d2 = NULL;
```

Here, the function **eye()** has been used with pointer variable *d2* to convert the data object *d2* into eye matrix.

Note: **eye()** function accepts only square matrices as inputs, otherwise it throws exception.

Creating Ones Matrix

Like Eye Matrix, Ones Matrix is equally important in matrix calculations. So we have developed a function called **ones()** under *DataObject* class to quickly convert any *data object* into *ones matrix*. Syntax for the function **ones()** is shown below.

dataObjectName.ones(dim 1,dim 2,...,dim n, dataType);

Return Type: void

```
1 ito::DataObject *d3 = new ito::DataObject();
2 d3->ones(2,3,4, ito::tFloat64);
3 std::cout << "2x3x4-ones matrix (double)" << *d3 << std::endl;
4 delete d3;
5 d3 = NULL;
```

Here, the function **ones()** has been used in line #2 with pointer variable *d3* to convert the data object *d3* into ones matrix of dimension 2x3x4.

Creating Zeros Matrix

We have developed a function called **zeros()** under *DataObject* class to quickly convert any *data object* into *zero matrix*. Syntax for the function **zeros()** is shown below.

dataObjectName.zeros(dataType);

dataObjectName.zeros(const size_t size, dataType);

dataObjectName.zeros(const size_t sizeY, const size_t sizeX, dataType);

dataObjectName.zeros(const size_t sizeZ, const size_t sizeY, const size_t sizeX, dataType);

*dataObjectName.zeros(const unsigned char dimensions, const size_t *sizes, dataType);*

Return Type: RetVal

As **zeros()** function is overloaded, there are more than one syntax shown above.

Note: The **RetVal** class is used for handling error management and return relative codes. More description on this class can be seen in [RetVal - Reference](#).

Following code explains the usage of **zeros()** function.

```
1 ito::DataObject *dObjZeros = new ito::DataObject();
2 dObjZeros->zeros(2,3,4,ito::tFloat64);
3 std::cout << "3x4x5-zeros matrix (double)" << *dObjZeros << std::endl;
4 delete dObjZeros;
5 dObjZeros = NULL;
```

Here, line number 2 shows the way to use zeros() function to convert data object dObjZeros into a zero matrix.

Adjusting ROI of a Data Object

This section will teach you about how to adjust Region of Interest (ROI) in any data object.

The following example code shows the way to adjust ROI with adjustROI() method and to locate ROI with locateROI() method.

```
1 //adjusting ROI of 6x7 data object.
2 ito::DataObject d6(6,7,ito::tInt16);
3 int roiLocate[] = {0,0,0,0}; //Empty Array to locate ROI of 2 dimensional data object d6.
4 d6.adjustROI(-2,0,-1,-4);
5 d6.locateROI(roiLocate);
6 std::cout << d6 << std::endl;
7 for(int i =0; i<4; i++)
8 {
9     std::cout << roiLocate[i] << std::endl;
10 }
```

Here, line #4 shows the use of adjustROI() function where negative parameters indicate that the ROI is shrinking in particular dimension. More detailed description of adjustROI() and locateROI() methods can be seen under [DataObject - Reference](#) document.

One can also pass an array as a parameter to this adjustROI() function describing the offset details as shown in the following code.

```
1 int matLimits2d[] = {-2,0,-1,-4};
2 d6.adjustROI(2,matLimits2d);
3 std::cout << d6 << std::endl;
```

Here, adjustROI() function is called with 2 parameters and as can be seen in line #1, the array matLimits2d[] contains the same offset values as passed in adjustROI() method in the previous example.

One can use this example to adjust ROI of data objects more than 2 dimensions as well as shown in later examples below.

The following code shows such an example to modify ROI of a 3 dimensional data object.

```
1 //adjusting ROI of 6x7x8 data object.
2 ito::DataObject d7(6,7,8,ito::tFloat32);
3 int matLimits3d[] = {-1,-2,0,-2,-3,-1};
4 int lims3d[] = {0,0,0,0,0,0}; //Empty Array to locate ROI of 3 dimensional data object d7.
5 d7.adjustROI(3,matLimits3d);
6 d7.locateROI(lims3d);
7 std::cout << d7 << std::endl;
8 for(int i = 0; i<5; i++)
9 {
10     std::cout << lims3d[i] << std::endl;
11 }
```

Here, a 3 dimensional data object d7 of dimensions 6x7x8 is declared in line #1 using the implementation #4 for data objects.

As can be seen in line #3, array `matLimits3d[]` of return type **int** contains required 6 offset values to adjust ROI of 3 dimensional data object d7. As shown in line #4, an empty array `lims3d[]` of *int* as return type is defined to locate the ROI of data object d7 using **locateROI()** function. Line #7 will print the resultant data object after being adjusted by **adjustROI()** method in line #5 and the for loop in line #8-11 will print these located offset values of the resultant ROI of d7.

Setting and Getting Axis Units

In this section, you will study about assigning and retrieving the axis units at each dimension of a data object. **setAxisUnit()** method is used to assign a unit to a particular axis (dimension) of a data object and is declared under **DataObject** class. Syntax for this method is given below.

dataObjectName.setAxisUnit(const unsigned int axisNum, const std::string &unit);

Return Type: int

This function returns 1 if the axis does not exists.

getAxisUnit() method is used to retrieve a unit of a particular axis and is also declared under **DataObject** class. The syntax for this method is shown below.

dataObjectName.getAxisUnit(const int axisNum, bool &validOperation);

Return Type: std::string

This method returns Null if the axisNum is out of the range.

Following example code explains both of these methods.

```
1 ito::DataObject d8(6,7,ito::tFloat32);
2 bool vop1, vop2 = 0;
3 d8.setAxisUnit(0, "cm");
4 d8.setAxisUnit(1, "cm");
5 std::string AxisUnit1 =d8.getAxisUnit(0,vop1); //Getting axis unit of 1st dimension of data object
6 std::string AxisUnit2 =d8.getAxisUnit(1,vop2); //Getting axis unit of 2nd dimension of data object
7 std::cout << "Axis Unit of 1st Dimension:" << AxisUnit1 << std::endl;
8 std::cout << "Axis Unit of 2nd Dimension:" << AxisUnit2 << std::endl;
```

Here, a 2 dimensional data object *d8* of dimensions 6x7 is declared in line #1. Line #2 declares boolean variables *vop1* and *vop2* to pass as parameters in `getAxisUnit()` method later. Line #3 and #4 sets the units for dimensions 1 and 2 of data object *d8* respectively.

These assigned axis units can be retrieved by `getAxisUnit()` method as shown in line #5 and #6. Line #7 and #8 prints these retrieved axis units of data object *d8*.

Setting and Getting Axis Scale

In this section, we will learn about setting and getting the scale values of particular Axis (dimension) of a data object. This can be done using `setAxisScale()` and `getAxisScale()` functions as shown below. These both functions are declared under *DataObject* class. Syntax for the `setAxisScale()` function is shown below:

DataObjectName.setAxisScale(const unsigned int axisNum, const double scale);

Return Type: int

In the same way, syntax for the **getAxisScale()** function is shown below.

DataObjectName.getAxisScale(const int axisNum)

Return Type: double

Following exemplary code can explain the usage of these functions in a better way.

```

1  ito::DataObject d9(6,5,3,ito::tInt16);
2  d9.setAxisScale(0,5);
3  d9.setAxisScale(1,-0.5);
4  d9.setAxisScale(2,3.24);
5  double AxisScale1 =d9.getAxisScale(0);
6  double AxisScale2 =d9.getAxisScale(1);
7  double AxisScale3 =d9.getAxisScale(2);
8  std::cout << "Axis 1 Scale:" << AxisScale1 << "Axis 2 Scale:" << AxisScale2 << "Axis 3 Scale:" <<

```

As shown in line #2-4, **setAxisScale()** function is used to assign scales of 5, -0.5 and 3.24 on Axis 0, 1 and 2 respectively of data object d9. Line #5-7 explains how the axis scales can be retrieved using the function **getAxisScale()**. Line #8 prints down these retrieved axis values at the end of this code snippet.

Copy data into an existing data object

Often, it is necessary to copy external data arrays into a region of interest, an entire plane or an entire data object. Of course this can be done using the methods from the accessing and assigning section above. However for the common case of copying 2D matrices into a two-dimensional region of interest or a 2D-plane of a data object, there exists the method **copyFromData2D**. The main requirement is, that the size of the data object exactly fits to the given size of the external array. Additionally, the types of the data must fit as well. Let's describe the method using two examples:

```

//example for method
ito::RetVal copyFromData2D(const _Tp *data, int sizeX, int sizeY);

//create 6 uint16 values
ito::uint16 arr[] = {1,2,3,4,5,6};

//create a 3x2x3, uint16 data object filled with 0
ito::DataObject b;
b.zeros(3,2,3,ito::tUInt16);

//get a slice of the second plane (careful: index (1,2) only selects one! plane)
ito::Range ranges2[] = {ito::Range(1,2),ito::Range::all(), ito::Range::all()};
ito::DataObject c = b.at(ranges2);

//let all values of c set to the values given by the first argument
//3,2 indicates the width and height of data, since arr is a one dimensional array.
ito::RetVal ret = c.copyFromData2D(arr,3,2);

//prints the resulting overall object b, where the second plane is [1,2,3;4,5,6]
std::cout << b << std::endl;

```

The second example allows to only copy a subpart of the given data block into the data object

```

//example for method
ito::RetVal copyFromData2D(const _Tp *data, int sizeX, int sizeY, \
    const int x0, const int y0, const int width, const int height);

//create 6 uint16 values
ito::uint16 arr[] = {1,2,3,4,5,6};

//create a 3x2x3, uint16 data object filled with 0
ito::DataObject b;
b.zeros(3,2,3,ito::tUInt16);

//let the values 2,3 and 5,6 be copied into the second and third column of the second plane
//of b.
ito::Range ranges2[] = {ito::Range(1,2),ito::Range::all(), ito::Range(1,3)};
ito::DataObject c = b.at(ranges2);
ito::RetVal ret = c.copyFromData2D(arr,3,2,1,0,2,2);

```

```
//prints the resulting overall object b, where the second plane is [0,2,3;0,5,6]
std::cout << b << std::endl;
```

The `copyFromData2D` is often used for copying camera data inside an internal or an externally given data object.

Operations on data objects

In this section, we will learn some basic operations which can be performed using data objects.

Adjugate of a data object

In many matrix calculations, there occurs a need to adjugate a matrix. Here, we also have a function `adj()` declared under `DataObject` class, which returns an adjugated matrix of the original data object. Syntax for this method is shown below.

DataObjectName.adj();

Return Type: `ito::DataObject*`

Following code snippet explains the use of `adj()` function.

```
1 ito::DataObject d10(6,5,3,ito::tComplex128);
2 d10.at<ito::complex128>(0,1,2) = cv::saturate_cast<ito::complex128>(ito::complex128(23.2,0));
3 d10.at<ito::complex128>(1,0,1) = cv::saturate_cast<ito::complex128>(ito::complex128(0,3));
4 d10.at<ito::complex128>(2,2,1) = cv::saturate_cast<ito::complex128>(ito::complex128(1234,-23.3));
5 ito::DataObject adjugatedDataObj = d10.adj();
6 std::cout << "The Adjugated data object:" << std::endl;
```

In the code above, a 6x5x3 data object `d10` of data type `ito::tComplex128` is created. Later on, in line #2-4, some complex values are assigned at data object elements (0,1,2), (1,0,1) and (2,2,1). In line #5, an adjugated matrix of data object `d10` is created using `adj()` function and stored in new data object called `adjugatedDataObj`. Line #6 prints out this adjugated matrix.

Transpose a data object

Transposing a data object is also one of the very important techniques in matrix calculations. With the use of `trans()` function, we can achieve a transposed matrix of the original data object. This function is also declared under `DataObject` class. Syntax for this function is shown below.

DataObjectName.trans()

Return Type: `ito::DataObject`

Following code snippet explains the usage of `trans()` function.

```
1 ito::DataObject d11(2,2,ito::tInt16);
2 int temp=0;
3 for(int i=0;i<2;i++)
4 {
5     for(int j=0;j<2;j++)
6     {
7         temp++;
8         d11.at<ito::int16>(i,j)= cv::saturate_cast<ito::int16>(temp);
9     }
10 }
11 ito::DataObject transDObj = d11.trans();
12 std::cout<< "The Transposed data object:" << std::endl;
```

In the code above, a 2x2 data object `d11` of type `ito::tInt16` is created. This data object `d11` is initiated by assigning different values to each element. In line #11, `trans()` function is used to transpose this data object `d11` and resulted transposed data object is stored in new data object `transDObj`, which gets printed out in line #12.

Note: Transposing the data object also transposes the axis related informations.

Squeeze a data object

The squeeze method of a data object is a convenient function to return a shallow copy of the data object where all dimensions with a size of 1 are eliminated. This does not have any impact on the underlying data of the data object but only the header information is changed. Therefore, a shallow copy is possible. The squeeze operations is often used, if another function requires for instance a two dimensional data object, but a three dimensional is given. Even, if only one plane of the three dimensional object is selected using any slicing operation (region of interest...), it is still three dimensional. After the squeeze-method however, a two dimensional object is obtained.

Example:

```
//3 x 3 x 2, float32 data object, all values = 0
ito::DataObject a(3,3,2, ito::tFloat32);
a = 0;

//create view or shallow copy of the first plane, using the range objects
ito::Range ranges[] = {ito::Range(1,2),ito::Range::all(), ito::Range::all()};
ito::DataObject secondPlane = a.at(ranges);

//secondPlane has a size of 1 x 3 x 2, this is squeezed
ito::DataObject squeezed = secondPlane.squeeze();

//squeezed has now a size of 3 x 2, set its first element to 2
squeezed.at<ito::float32>(0,0) = 2;

//print out the original object
std::cout << a << std::endl;

//the result is:
/* [[0,0,0;0,0,0];[2,0,0;0,0,0];[0,0,0;0,0,0]]
```

Note: The last two dimensions belonging to planes are never squeezed, since this would require a full recreation of the data object including a deep copy of the data.

Basic operators with data objects

(Same syntax can be used for other operators like '+', '-', '=', '+=', '=-', div, cross multiplication (!=), << (shift left), >> (shift right)) For the sake of simplicity, some arithmetic operators are overloaded to work upon data objects easily. In this section, such operators to work upon data objects are discussed in details with example codes and syntaxes. Let us start with basic Add “+” operator. Following is one example shown to add two data objects with “+” operator.

```
1 ito::DataObject d12(2,2,ito::tInt16);
2 ito::DataObject d13(2,2,ito::tInt16);
3 ito::DataObject d14(2,2,ito::tInt16);
4 d12= cv::saturate_cast<ito::int16>(2);
5 d13= cv::saturate_cast<ito::int16>(2);
6 d14= d12 + d13;
7 std::cout << "Addition of two matrix is:" << d14 << std::endl;
```

Here, two data objects *d12* and *d13* are added element-wise and the resultant data object is stored in *d14*. In the same way many other arithmetic, compare or bitwise operators can work with data objects. In the following, many operators are introduced that are overloaded for working with data objects as argument.

Arithmetic operators addition and subtraction

Using the `+` or `-` operator, two data objects of the same size and same type can be added/subtracted by means of an element-wise addition/subtraction:

```
ito::DataObject mat3 = mat1 + mat2;
mat3 = mat1 - mat2;
```

The same operations can also be done inplace, such that one data object is added to or subtracted from another data object, that is directly modified:

```
mat2 += mat1;
mat2 -= mat1;
```

multiplication and division

At first, the star-operator `*` is overloaded, such it is possible to multiply a constant factor to all values of the given data object. This can also be done inplace:

```
ito::DataObject mat2 = mat1 * 2.0;
mat1 *= 2.0; //inplace
```

When using the star-operator with two data objects, one needs to know that this indicates a matrix-multiplication in the mathematical sense, hence a $m \times n$ matrix multiplied with a $n \times k$ matrix results in a $m \times k$ matrix:

```
ito::DataObject mat3 = mat1 * mat2;
```

Note: This operation is only defined for *float32* and *float64* datatypes.

An element-wise multiplication of two data objects of same size and type is obtained by using the **mul** method of the first data object:

```
ito::DataObject mat3 = mat1.mul(mat2);
```

The element-wise division is finally obtained by the corresponding **div** method:

```
ito::DataObject mat3 = mat1.div(mat2);
```

Note: The `“div”` operator can not be used to calculate inverse matrix.

Comparison operators The comparison operators can be used to element-wisely compare the elements of two data objects of same size and type. The result of all comparisons is a data object with the same size than the compared objects but with the fixed type *uint8*. Depending on the comparison, this resulting matrix contains a value of **0** or **1**. The comparison operators are not supported for the **int8** and **int32** data types (OpenCV restriction). Available operators are:

```
ito::DataObject result = (mat1 == mat2); //equal to
ito::DataObject result = (mat1 != mat2); //unequal to
ito::DataObject result = (mat1 < mat2); //lower than
mat1 <= mat2; //lower or equal than
mat1 > mat2; //bigger than
mat1 >= mat2; //bigger or equal than
```

Shift operators Shift operators play a significant role during some arithmetic operations (i.e. division/multiplication by 2), bit level calculations, etc. The shift operators are only defined for signed or unsigned fixed point data types. A left shift can either return the resulting data object or can be done in-place. The left shift moves every element in a data object by *x* places to the left such that the resulting number is multiplied by two to the power of *x*:

```

unsigned int x = 2;
ito::DataObject result = mat1 << x; //left-shift
mat <<= x; //in-place

```

The right-shift is similar but it moves the binary number by x places to the right, hence, the number is divided by two to the power of x .

```

unsigned int x = 2;
ito::DataObject result = mat1 >> x; //right-shift
mat >>= x; //in-place

```

Bitwise operations The bitwise operators are also executed as an element-wise operation and are defined for all data types. The participating data objects must again have the same size and type. Examples are:

```

ito::DataObject result = mat1 & mat2; //bitwise and-combination
result = mat1 | mat2; //bitwise or-combination
result = mat1 ^ mat2; //bitwise nor-combination

//inplace:
mat1 &= mat2;
mat1 |= mat2;
mat1 ^= mat2;

```

Combination of different operators Different operators (explained above) can be used in different possible combinations with data objects for speedy calculations.

Some of the examples of such combinations are shown below.

```

1 d4 = d1 + d2 - d3;
2 d3 = (d1 | d2).div((d1 << 1) + d1);
3 d4= (d1 & d2).mul(d3 >> 2);

```

With the close observation of the example statements above, one can get an idea on how to use these operators in different combinations to get the desired operation done.

Note: For a full reference of the class **DataObject** see [DataObject - Reference](#).

DataObject - Reference

class **ito::DataObject**

dataObject contains a n-dimensional matrix

The n-dimensional matrix can have different element types. Recently the following types are supported: int8, uint8, int16, uint16, int32, uint32, float32, float64 (=> double), complex64 (2x float32), complex128 (2x float64)

In order to handle huge matrices, the data object can divide one matrix into subparts in memory. Each subpart (called matrix-plane) is two-dimensional and covers data of the last two dimensions. Each of these matrix-planes is of type `cv::Mat_<type>` and can be used with every operator given by the openCV-framework (version 2.3.1 or higher).

We assume to have a n-dimensional matrix A, where each dimension has its size s_i , hence $A=[s_1, s_2, ..., s_{(n-2)}, s_{(n-1)}, s_n]$

Hence, in total there are $s_1 * s_2 * ... * s_{(n-2)}$ different matrix-planes, which are all accessible by the member `m_data`, which is a `std::vector` of the general type `int*`. This type has to be casted to the specific `cv::Mat_<...>` when one matrix-plane has to be accessed. Sometimes it is also possible to simply cast to `cv::Mat`.

In order to make the data object compatible to continuously organized data structures, like numpy-arrays, it is also possible to have all matrix-planes in one data-block in memory. Then the continuous-flag will be

set and the whole data block can be accessed by taking the pointer given by `m_data[0]`. Nevertheless, the indicated data structure with the two-dimensional sub-matrix-planes is still existing, hence, the pointer to each matrix-planes points to the entry point of its matrix-planes lying within the huge data block.

The data organization is equal to the one of open-cv, hence, two-dimensional matrices are stored row-by-row (C-style)...

The real size of each dimension is stored in the vector `m_ose`. Since it is possible to set a n-dimensional region of interest (ROI) to each matrix, the virtual dimensions, which will be delivered if the user asks for the matrix size, are stored in the member vector `m_size`.

Concept to handle templated and non-templated methods _____

According to openCV, the class `dataObject` is not templated, because there are some structures in the entire itom-framework which does not support any templating concept, like the plugin-handling or communication with external dll-functions. Additionally the signal-slot-design of the Qt-framework does not accept templated parameters beside some standard-objects. Therefore the element-data-type is set by the integer-member `m_type`. The transformation between the real data type and the integer number is coded several times within the whole framework and can be accessed by the enumeration `tDataType` in `typeDefs.h`. Since templating has got many advantages concerning low-level calculation, we adapted the transformation-process which is used by openCV:

1. define a templated helper-method in the following form:

```
template<typename _Tp> returnType 'MethodName'Func(Parameters1)
```

2. define the following two lines of code: `typedef returnType (*t'MethodName'Func)(Parameters1);`
`MAKEFUNCLIST('MethodName'Func);`

3. define the method, accessed for example as public-method of `dataObject` *RetVal DataObject::'PublicMethodName'(Parameters2)* { ... `fList'MethodName'Func[getType()]`(Parameters1); ... return ... }

— By the macro `MAKEFUNCLIST` a list `fList'MethodName'Func` is generated with each entry being a function pointer to the specific templated version of `'MethodName'Func`. The specific method is accessed by using `getType()` of `dataObject`. Hence it is important to keep the element-data-types and their order consistent for the whole itom-project.

Public Functions

DataObject (void)

constructor for empty data object

no data will be allocated, the number of elements and dimensions is set to zero

DataObject (const int *size*, const int *type*)

constructor for one-dimensional data object. The data is newly allocated and arbitrarily filled.

In fact, by this constructor a two-dimensional matrix with dimension 1 x *size* will be created. the `owndata`-flag is set to true, the `continuously`-flag, too (since only one matrix-plane will be created)

See

create, `tDataType`

Parameters

- `size` - is the number of elements
- `type` - is the data-type of each element (use type of enumeration `tDataType`)

DataObject (const int *sizeY*, const int *sizeX*, const int *type*)

constructor for two-dimensional data object. The data is newly allocated and arbitrarily filled.

the `owndata`-flag is set to true, the `continuously`-flag, too (since only one matrix-plane will be created)

See

create, tDataType

Parameters

- `sizeY` - is the number of rows in each matrix-plane
- `sizeX` - is the number of columns in each matrix-plane
- `type` - is the data-type of each element (use type of enumeration `tDataType`)

DataObject (const int *sizeZ*, const int *sizeY*, const int *sizeX*, const int *type*, const unsigned char *continuous* = 0)

constructor for three-dimensional data object. The data is newly allocated and arbitrarily filled.

the `owndata`-flag is set to true

See

create, tDataType

Parameters

- `sizeZ` - is the number of images in the z-direction
- `sizeY` - is the number of rows in each matrix-plane
- `sizeX` - is the number of columns in each matrix-plane
- `type` - is the data-type of each element (use type of enumeration `tDataType`)
- `continuous` - indicates whether all matrix-planes should continuously lie in memory (1) or not (0) (default: 0)

DataObject (const int *sizeZ*, const int *sizeY*, const int *sizeX*, const int *type*, const uchar * *continuousDataPtr*, const int * *steps* = NULL)

constructor for 3-dimensional data object which uses the data given by the `continuousDataPtr`.

In case of the `continuousDataPtr`, the `owndata`-flag is set to false, hence this `dataObj` will not delete the data. Additionally the `continuous`-flag is set to true.

See

create, tDataType

Parameters

- `sizeZ` - is the number of images in the z-direction
- `sizeY` - is the number of rows in each matrix-plane
- `sizeX` - is the number of columns in each matrix-plane
- `type` - is the data-type of each element (use type of enumeration `tDataType`)
- `*continuousDataPtr` - points to the first element of a continuous data block of the specific data type
- `*steps` - may be NULL, if the data in `continuousDataPtr` should be taken continuously, hence the ROI is the whole matrix, else this is a vector with three elements, where each elements indicates the number of bytes one has to move in order to get from one element to the next one in the same dimension. Hence, the last element in this vector is equal to the size of one single element (in bytes)

DataObject (const MSize & *sizes*, const int *type*, const unsigned char *continuous* = 0)

constructor for data object with given dimension. The data is newly allocated and arbitrarily filled.

the `owndata`-flag is set to true

See

create, tDataType

Parameters

- `sizes` -
- `type` - is the data-type of each element (use type of enumeration tDataType)
- `continuous` - indicates whether all matrix-planes should continuously lie in memory (1) or not (0) (default: 0)

DataObject (const unsigned char *dimensions*, const int * *sizes*, const int *type*, const unsigned char *continuous* = 0)

constructor for data object with given dimension. The data is newly allocated and arbitrarily filled.

the owndata-flag is set to true

See

create, tDataType

Parameters

- `dimensions` - indicates the total number of dimensions
- `*sizes` - is a vector of size 'dimensions', where each element gives the size (not osize) of the specific dimension
- `type` - is the data-type of each element (use type of enumeration tDataType)
- `continuous` - indicates whether all matrix-planes should continuously lie in memory (1) or not (0) (default: 0)

DataObject (const unsigned char *dimensions*, const int * *sizes*, const int *type*, const uchar * *continuousDataPtr*, const int * *steps* = NULL)

constructor for data object which uses the data given by the continuousDataPtr.

In case of the continuousDataPtr, the owndata-flag is set to false, hence this dataObj will not delete the data. Additionally the continuous-flag is set to true.

See

create, ito::tDataType

Parameters

- `dimensions` - indicates the total number of dimensions
- `*sizes` - is a vector of size 'dimensions', where each element gives the size (not osize) of the specific dimension
- `type` - is the data-type of each element (use type of enumeration tDataType)
- `*continuousDataPtr` - points to the first element of a continuous data block of the specific data type
- `*steps` - may be NULL, if the data in continuousDataPtr should be taken continuously, hence the ROI is the whole matrix, else this is a vector of size 'dimensions', where each elements indicates the number of bytes one has to move in order to get from one element to the next one in the same dimension. Hence, the last element in this vector is equal to the size of one single element (in bytes)

DataObject (const *DataObject* & *copyConstr*)

copy constructor for data object

copy constructor

creates a data object with respect to the given data object. The header information is completely copied, while the data is a shallow copy. The lock of the new data object is unlocked while the lock for the common data block is taken from the current lock status of the given data object.

Parameters

- `©Constr` - is the data object, which will be copied

~DataObject (void)
destructor

reference pointer of data is decremented and if <0, data will be deleted if `owndata-flag` is true. Additionally the allocated memory for header information will be deleted, too.

See

`freeData`

double **getValueOffset** () const
< Function return the offset of the values stored within the `dataObject`
Function return the scaling of values stored within the `dataObject`

double **getValueScale** () const
Function return the unit description for the values stored within the `dataObject`.

const std::string **getValueUnit** () const
Function return the description for the values stored within the `dataObject`, if `tagSpace` does not exist, NULL is returned.

std::string **getValueDescription** () const
Function return the axis-offset for the existing axis specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.

double **getAxisOffset** (const int *axisNum*) const
Function returns the axis-description for the exist axis specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.

double **getAxisScale** (const int *axisNum*) const
Function returns the axis-unit-description for the exist axis specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.
< Function returns the axis-description for the exist axis specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.

const std::string **getAxisUnit** (const int *axisNum*, bool & *validOperation*) const
Function returns the axis-description for the exist specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.
< Function returns the axis-unit-description for the exist axis specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.

std::string **getAxisDescription** (const int *axisNum*, bool & *validOperation*) const
< Function returns the axis-description for the exist specified by `axisNum`. If `axisNum` is out of dimension range it returns NULL.

bool **getTagByIndex** (const int *tagNumber*, std::string & *key*, `DataObjectTagType` & *value*) const
Function returns the string-value for 'key' identified by `tagNumber`. If key in the `TagMap` do not exist NULL is returned.

std::string **getTagKey** (const int *tagNumber*, bool & *validOperation*) const
Function returns the number of elements in the `Tags-Maps`.
< Function returns the string-value for 'key' identified by `tagNumber`. If key in the `TagMap` do not exist NULL is returned

int **getTagListSize** () const
 Function to set the string-value of the value unit, return 1 if values does not exist.
 < Function returns the number of elements in the Tags-Maps

int **setValueUnit** (const std::string & *unit*)
 Function to set the string-value of the value description, return 1 if values does not exist.
 < Function to set the string-value of the value unit, return 1 if values does not exist

int **setValueDescription** (const std::string & *description*)
 < Function to set the string-value of the value description, return 1 if values does not exist

int **setAxisOffset** (const unsigned int *axisNum*, const double *offset*)
 < Function to set the offset of the specified axis, return 1 if axis does not exist

int **setAxisScale** (const unsigned int *axisNum*, const double *scale*)
 < Function to set the scale of the specified axis, return 1 if axis does not exist or scale is 0.0.

int **setAxisUnit** (const unsigned int *axisNum*, const std::string & *unit*)
 < Function to set the unit (string value) of the specified axis, return 1 if axis does not exist

int **setAxisDescription** (const unsigned int *axisNum*, const std::string & *description*)
 < Function to set the description (string value) of the specified axis, return 1 if axis does not exist

int **setTag** (const std::string & *key*, const DataObjectTagType & *value*)
 < Function to set the string value of the specified tag, if the tag do not exist, it will be added automatically, return 1 if tagspace does not exist

bool **existTag** (const std::string & *key*) const
 < Function to check whether tag exist or not

bool **deleteTag** (const std::string & *key*)
 < Function deletes specified tag. If tag do not exist, return value is 1 else returnvalue is 0

int **addToProtocol** (const std::string & *value*)
 < Function adds value to the protocol-tag. If this object is an ROI, the ROI-coordinates are added. If string do not end with an , is added.

double **getPhysToPix** (const unsigned int *dim*, const double *phys*, bool & *isInsideImage*) const
 Function returns the not rounded pixel index of a physical coordinate.

Function returns the not rounded pixel index of a physical coordinate Function returns the not rounded pixel index of a physical coordinate (Unit-Coordinate = (px-Coordinate - Offset)* Scale). If the pixel is outside of the image, the isInsideImage-flag is set to false else it is set to true. To avoid memory access-error, the returnvalue is clipped within the range of the image ([0...imagesize-1])

Return

(double)(pix / AxisScale + AxisOffset) & [0..imagesize-1]

Parameters

- *dim* - Axis-dimension for which the physical coordinate is calculated
- *pix* - Pixel-index as double
- *isInsideImage* - flag which is set to true if coordinate is within range of the image.

double **getPhysToPix** (const unsigned int *dim*, const double *phys*) const
 Function returns the not rounded pixel index of a physical coordinate.

Function returns the not rounded pixel index of a physical coordinate Function returns the not rounded pixel index of a physical coordinate (Unit-Coordinate = (px-Coordinate - Offset)* Scale). To avoid memory access-error, the return value is clipped within the range of the image ([0...imagesize-1])

Return

(double)(pix / AxisScale + AxisOffset) & [0..imagesize-1]

Parameters

- `dim` - Axis-dimension for which the physical coordinate is calculated
- `pix` - Pixel-index as double

int **getPhysToPix2D** (const double *physY*, double & *tPxY*, bool & *isInsideImageY*, const double *physX*, double & *tPxX*, bool & *isInsideImageX*) const
 Function returns the not rounded pixel index of a physical coordinate.

double **getPixToPhys** (const unsigned int *dim*, const double *pix*, bool & *isInsideImage*) const
 Function returns the physical coordinate of a pixel.

Function returns the physical coordinate of a pixel
 Function returns the physical coordinate of a pixel index (Unit-Coordinate = (px-Coordinate - Offset)* Scale). If the pixel is outside of the image, the *isInsideImage*-flag is set to false else it is set to true.

Return

(double)(`pix` - AxisOffset)* AxisScale)

Parameters

- `dim` - Axis-dimension for which the physical coordinate is calculated
- `pix` - Pixel-index as double
- `isInsideImage` - flag which is set to true if coordinate is within range of the image.

double **getPixToPhys** (const unsigned int *dim*, const double *pix*) const
 Function returns the physical coordinate of a pixel.

Function returns the physical coordinate of a pixel
 Function returns the physical coordinate of a pixel index (Unit-Coordinate = (px-Coordinate - Offset)* Scale).

Return

(double)(`pix` - AxisOffset)* AxisScale)

Parameters

- `dim` - Axis-dimension for which the physical coordinate is calculated
- `pix` - Pixel-index as double

RetVal **setXYRotationalMatrix** (double *r11*, double *r12*, double *r13*, double *r21*, double *r22*, double *r23*, double *r31*, double *r32*, double *r33*)
 Function to access (set) the rotational matrix by each element.

Return

ito::retOk || ito::retError

Parameters

- `r11` - Upper left element
- `r12` - Upper middle element
- `r13` - Upper righth element
- `r21` - Middle left element
- `r22` - Middle middle element
- `r23` - Middle righth element
- `r31` - Lower left element
- `r32` - Lower middle element
- `r33` - Lower righth element

RetVal **getXYRotationalMatrix** (double & *r11*, double & *r12*, double & *r13*, double & *r21*,
double & *r22*, double & *r23*, double & *r31*, double & *r32*,
double & *r33*) const

Function to access (get) the rotational matrix by each element.

Return

ito::retOk || ito::retError

Parameters

- *r11* - Upper left element
- *r12* - Upper middle element
- *r13* - Upper right element
- *r21* - Middle left element
- *r22* - Middle middle element
- *r23* - Middle right element
- *r31* - Lower left element
- *r32* - Lower middle element
- *r33* - Lower right element

RetVal **copyTagMapTo** (*DataObject* & *rhs*) const

Deep copies the tagmap with all entries to rhs object

this function makes a deepcopy of the tags map to rhs object from this object.

Return

retOk

See

DataObjectTags

Parameters

- &*rhs* - is the matrix where the map is copied to. The old map of this object is cleared first

RetVal **copyAxisTagsTo** (*DataObject* & *rhs*) const

Deep copies the axistags to rhs object

this function makes a deepcopy of the axis and value metadata from this object to rhs object. It copies

Return

retOk

See

DataObjectTags

Parameters

- &*rhs* - is the matrix where the map is copied from. The old map of this object is cleared first

int **getDims** (void) const

< returns the number of dimensions returns the element data type in form of its type-number

int **getType** (void) const

returns if the data in the first n-2 dimensions is stored within one entire block in memory (true), else (false)

char **getContinuous** (void) const

returns if the data object is owner of the data, hence, the data will be deleted by this data object, if nobody else is using the data any more

char **getOwnData** (void) const

returns the real plane index of cv::Mat array returned by [get_mdata\(\)](#) for a given plane number considering a possible roi. Use this method if you already know the total number of planes within the roi.

int **seekMat** (const int *matNum*, const int *numMats*) const

returns the index vector-index of m_data which corresponds to the given zero-based two-dimensional matrix-index

returns the real plane index of cv::Mat array returned by [get_mdata\(\)](#) for a given plane number considering a possible roi. This method internally calculates the number of planes within the roi using [getNumPlanes](#).

Since there might be a difference between the “real” matrix size in memory and the virtual size which is set by subslicing a matrix and hence setting any ROI, this method transforms a desired matrix-plane index to the real index in memory of the data-vector m_data

Return

real vector-index for the desired matrix-plane or 0 if matNum >= numMats.

Parameters

- *matNum* - zero-based matrix-plane-index, considering the virtual matrix size (ROI), $0 \leq \text{matNum} < \text{getNumPlanes}$
- *numMats* - total number of matrix-planes, lying within the ROI

int **seekMat** (const int *matNum*) const

returns the index vector-index of m_data which corresponds to the given zero-based two-dimensional matrix-index

returns the number of planes of this data object (considering a possible ROI). This method simply calls [getNumPlanes](#) and is only there for historical reasons.

Since there might be a difference between the “real” matrix size in memory and the virtual size which is set by subslicing a matrix and hence setting any ROI, this method transforms a desired matrix-plane index to the real index in memory of the data-vector m_data

Return

real vector-index for the desired matrix-plane

See

[seekMat](#)

[getNumPlanes](#)

Parameters

- *matNum* - zero-based matrix-plane considering the virtual matrix size (ROI), $0 \leq \text{matNum} < \text{getNumPlanes}$

int **calcNumMats** (void) const

calculates numbers of single opencv matrices which are part of the ROI which has previously been set.

Return

0 if empty range or empty matrix, 1 if two dimensional, else product of sizes of all dimensions besides the last two ones.

See

[getNumPlanes](#)

int **getNumPlanes** (void) const

calculates numbers of single opencv matrices which are part of the ROI which has previously been set.

This method replaces calcNumMats due to its more consistent method name.

Return

0 if empty range or empty matrix, 1 if two dimensional, else product of sizes of all dimensions besides the last two ones.

cv::Mat * **getCvPlaneMat** (const int *planeIndex*)

returns pointer to cv::Mat plane with given index considering a possible roi.

returns the pointer to the underlying cv::Mat that represents the plane with given planeIndex of the entire data object.

This command is equivalent to *get_mdata()*[seekMat(planeIndex)] but checks for out-of-range errors.

Return

pointer to the cv::Mat plane or NULL if planeIndex is out of range

See

seekMat

get_mdata, *getContinuousCvPlaneMat*

Parameters

- *planeIndex* - is the zero-based index of the requested plane within the current ROI of the data object

const cv::Mat * **getCvPlaneMat** (const int *planeIndex*) const

returns pointer to cv::Mat plane with given index considering a possible roi.

returns the pointer to the underlying cv::Mat that represents the plane with given planeIndex of the entire data object.

This command is equivalent to *get_mdata()*[seekMat(planeIndex)] but checks for out-of-range errors.

Return

pointer to the cv::Mat plane or NULL if planeIndex is out of range

See

seekMat

get_mdata, *getContinuousCvPlaneMat*

Parameters

- *planeIndex* - is the zero-based index of the requested plane within the current ROI of the data object

const cv::Mat **getContinuousCvPlaneMat** (const int *planeIndex*) const

returns a shallow or deep copy of a cv::Mat plane with given index. If the current plane is not continuous (due to a roi), a cloned, continuous matrix is returned, else a shallow copy.

Return

shallow copy or clone of desired plane, depending if the plane is continuous (no roi set in plane dimensions) or not.

See

seekMat

get_mdata, *getCvPlaneMat*

Parameters

- `planeIndex` - is the zero-based index of the requested plane within the current ROI of the data object

`cv::Mat** get_mdata (void)`

returns array of pointers to `cv::_Mat`-matrices (planes) of the data object.

The returned array of matrices contains all matrices of this object, including the matrices that may lie outside of a possible region of interest. In order to access the *i*-th plane considering any roi, use `getCvPlaneMat` or calculate the right accessing index using `seekMat`.

Return

pointer to vector of matrices

Remark

the returned type is an array of `cv::Mat*`, you should cast it to the appropriate type (e.g. `cv::_Mat<int8>`)

See

[seekMat](#), [getCvPlaneMat](#)

`const cv::Mat** get_mdata (void) const`

returns constant array of pointers to `cv::_Mat`-matrices (planes) of the data object

The returned array of matrices contains all matrices of this object, including the matrices that may lie outside of a possible region of interest. In order to access the *i*-th plane considering any roi, use `getCvPlaneMat` or calculate the right accessing index using `seekMat`.

Return

pointer to vector of matrices

Remark

the returned type is a const array of `cv::Mat*`, you should cast it to the appropriate type (e.g. `cv::_Mat<int8>`)

See

[seekMat](#), [getCvPlaneMat](#)

`MSize getSize (void)`

returns the size-member. `m_size` fits to the physical organization of data in memory.

Return

size-member of type `MSize`

`const MSize getSize (void) const`

returns the size-member. This member does not consider the transpose flag, hence, `m_size` fits to the physical organization of data in memory.

Return

size-member of type `MSize`

`int getSize (int index) const`

gets the size of the given dimension (this is the size within the ROI)

Return

size or -1 if index is out of boundaries

Parameters

- `index` - is the specific zero-based dimension-index whose size is requested

int **getOriginalSize** (int *index*) const

gets the original size of the given dimension (this is the size without considering any ROI)

Return

size or -1 if index is out of boundaries

Parameters

- *index* - is the specific zero-based dimension-index whose size is requested

int **getTotal** () const

gets total number of elements within the data object's ROI

Return

number of elements

See

getDims, getSize

int **getOriginalTotal** () const

gets total number of elements of the whole data object

Return

number of elements

See

getDims, getSize

RetVal **copyTo** (*DataObject* & *rhs*, unsigned char *regionOnly* = 0) const

high-level, non-templated method to deeply copy the data of this matrix to another matrix *rhs*

deeply copies the data of this data object to the given *rhs-dataObject*, whose existing data will be deleted first.

Return

retOk

See

CopyToFunc

Parameters

- *&rhs* - is the matrix where the data is copied to. The old data of *rhs* is deleted first
- *regionOnlyif* - true, only the data of the ROI in *lhs* is copied, hence, the org-size of *rhs* corresponds to the ROI-size of *lhs*, else the whole data block is copied and the ROI of *rhs* is set to the ROI of *lhs*

RetVal **convertTo** (*DataObject* & *rhs*, const int *type*, const double *alpha* = 1, const double *beta* = 0) const

high-level, non-templated matrix conversion

Convertes an array to another data type with optional scaling ($\alpha * \text{value} + \beta$)

Every element of the source matrix is converted to a new, given type. Additionally a floating-point scaling and offset parameter is possible.

Return

retOk

See

fListConvertToFunc

Parameters

- `&rhs` - is the destination data object, whose memory is firstly deleted, then newly allocated
- `type` - is the type-number of the destination element
- `alpha` - scaling factor (default: 1.0)
- `beta` - offset value (default: 0.0)

Exceptions

- `cv::Exception` - if cast failed, e.g. if cast not possible or types unknown

RetVal **setTo** (const int8 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type uint8 or empty data object if no mask should be considered (default)

RetVal **setTo** (const uint8 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type uint8 or empty data object if no mask should be considered (default)

RetVal **setTo** (const int16 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type uint8 or empty data object if no mask should be considered (default)

RetVal **setTo** (const uint16 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- *assigned* - scalar converted to the actual array type
- *mask* - Operation mask of the same size as *this and type uint8 or empty data object if no mask should be considered (default)

RetVal **setTo** (const int32 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- *assigned* - scalar converted to the actual array type
- *mask* - Operation mask of the same size as *this and type uint8 or empty data object if no mask should be considered (default)

RetVal **setTo** (const uint32 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- *assigned* - scalar converted to the actual array type
- *mask* - Operation mask of the same size as *this and type uint8 or empty data object if no mask should be considered (default)

RetVal **setTo** (const float32 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if uint8 mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type `uint8` or empty data object if no mask should be considered (default)

RetVal **setTo** (const float64 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if `uint8` mask is given) of the array elements to the specified value.

Return

`retError` in case of error

See

`AssignScalarValue`

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type `uint8` or empty data object if no mask should be considered (default)

RetVal **setTo** (const complex64 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if `uint8` mask is given) of the array elements to the specified value.

Return

`retError` in case of error

See

`AssignScalarValue`

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type `uint8` or empty data object if no mask should be considered (default)

RetVal **setTo** (const complex128 & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if `uint8` mask is given) of the array elements to the specified value.

Return

`retError` in case of error

See

`AssignScalarValue`

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type `uint8` or empty data object if no mask should be considered (default)

RetVal **setTo** (const `ito::Rgb32` & *value*, const *DataObject* & *mask* = *DataObject*())

Sets all or some of the array elements to the specific value.

Sets all or some (if `uint8` mask is given) of the array elements to the specified value.

Return

retError in case of error

See

AssignScalarValue

Parameters

- `assigned` - scalar converted to the actual array type
- `mask` - Operation mask of the same size as `*this` and type `uint8` or empty data object if no mask should be considered (default)

RetVal **deepCopyPartial** (*DataObject* & *copyTo*)

copy all values of this data object to the copyTo data object. The copyTo-data object must be allocated and have the same type and size (of its roi) than this data object.

high-level, non-templated method. Deeply copies data of this data object which is within its ROI to the ROI of rhs.

Return

retOk

See

DeepCopyPartialFunc

Parameters

- `&rhs` - is the right-handed data object, where data is copied to.

Exceptions

- `cv::Exception(CV_StsAssert)` - if sizes or type of both matrices are not equal

DObjIterator **begin** ()

Returns the matrix iterator and sets it to the first matrix element.

returns iterator to the first item in the data object array

Return

iterator

See

DObjIterator

DObjIterator **end** ()

Returns the matrix iterator and sets it to the after-last matrix element.

returns iterator to the end value of this data object array

The end value is the first item outside of the data object array.

Return

iterator

See

DObjIterator

DObjConstIterator **constBegin** () const

Returns the matrix read-only iterator and sets it to the first matrix element.

returns constant iterator to the first item in the data object array

Return

iterator

See

DObjConstIterator

DObjConstIterator **constEnd** () const

Returns the matrix read-only iterator and sets it to the after-last matrix element.

returns constant iterator to the end value of this data object array

The end value is the first item outside of the data object array.

Return

iterator

See

DObjConstIterator

DataObject & **operator=** (const cv::Mat & *rhs*)

assign-operator which creates a two-dimensional data object as a shallow copy of a two dimensional cv::Mat object.

shallow-copy means, that the header information of this data-object is physically created at the hard disk, while the data is shared with the original cv::Mat.

Return

this data object

See

create

Parameters

- *&rhs* - is the cv::Mat where the shallow copy is taken from. At first, the existing data of this object is freed.

Exceptions

- *cv::Exception* - if *rhs* is not two-dimensional or data type has no compatible data type of *DataObject*.

DataObject & **operator=** (const *DataObject* & *rhs*)

assign-operator which makes a shallow-copy of the *rhs* data object and stores it in this data object

shallow-copy means, that the header information of the *rhs* data-object is physically copied to this-*DataObject* while the data is shared, hence, only its reference counter is incremented.

The previous array covered by this data object is completely released before assigning the new *rhs* data object. In order to deeply copy the values from one object into another pre-allocated object use the method *deepCopyPartial*.

Return

this data object

See

CopyMatFunc, *deepCopyPartial*

Parameters

- *&rhs* - is the data object where the shallow copy is taken from. At first, the existing data of this object is freed.

Exceptions

- `cv::Exception` - if lock state of both objects is not equal. Please make sure, that both lock states are equal

DataObject & **operator=** (const int8 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const uint8 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const int16 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const uint16 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const int32 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const uint32 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const float32 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const float64 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- *value* - is the scalar assignment value

DataObject & **operator=** (const complex64 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- `value` - is the scalar assignment value

DataObject & **operator=** (const complex128 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- `value` - is the scalar assignment value

DataObject & **operator=** (const ito::Rgb32 & *value*)

Every data element in this data object is set to the given value.

sets all elements of the data object to the given value. Value is cast to the data object's type

Return

modified data object

See

AssignScalarValue

Parameters

- `value` - is the scalar assignment value

DataObject & **operator+=** (const *DataObject* & *rhs*)

high-level, non-templated arithmetic operator for element-wise addition of values of given data object to this data object

Return

this data object

See

AddFunc

Parameters

- `&rhs` - is the data object whose elements will be added to this data object

Exceptions

- `cv::Exception` - if both data objects don't have the same size or type

DataObject **operator+** (const *DataObject* & *rhs*)

high-level, non-templated arithmetic operator for element-wise addition of values of two given data objects

Return

new resulting data object

See

AddFunc

Parameters

- `&rhs` - is the data object whose elements will be added to this data object

Exceptions

- `cv::Exception` - if both data objects don't have the same size or type

DataObject & **operator--** (const *DataObject* & *rhs*)

high-level, non-templated arithmetic operator for element-wise subtraction of values of given data object from values of this data object

Return

this data object

See

SubFunc

Parameters

- `&rhs` - is the data object whose elements will be subtracted from this data object

Exceptions

- `cv::Exception` - if both data objects don't have the same size or type

DataObject **operator-** (const *DataObject* & *rhs*)

high-level, non-templated arithmetic operator for element-wise subtraction of values of given data object from values of this data object

Return

new resulting data object

See

SubFunc

Parameters

- `&rhs` - is the data object whose elements will be subtracted from this data object

Exceptions

- `cv::Exception` - if both data objects don't have the same size or type

DataObject & **operator*=** (const *DataObject* & *rhs*)

brief description

DataObject & **operator*=** (const float64 & *factor*)

high-level method which multiplies every element in this data object by a given floating-point factor

See

OpScalarMulFunc

Parameters

- `factor` -

DataObject **operator*** (const *DataObject* & *rhs*)

brief description

DataObject **operator*** (const float64 & *factor*)

high-level method which multiplies every element in this data object by a given floating-point factor. The result matrix is returned as a new matrix.

See

operator *, OpScalarMulFunc

Parameters

- `factor` -

DataObject **operator<** (*DataObject* & *rhs*)

compare operator, compares for “lower than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `&rhs` - is the data object with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator>** (*DataObject* & *rhs*)

compare operator, compares for “bigger than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `&rhs` - is the data object with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator<=** (*DataObject* & *rhs*)

compare operator, compares for “lower or equal than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `&rhs` - is the data object with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator>=** (*DataObject* & *rhs*)

compare operator, compares for “bigger or equal than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `&rhs` - is the data object with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator==** (*DataObject* & *rhs*)

compare operator, compares for "equal to"

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `&rhs` - is the data object with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator!=** (*DataObject* & *rhs*)

compare operator, compares for "unequal to"

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `&rhs` - is the data object with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator<** (const float64 & *value*)

compare operator, compares for "lower than"

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `value` - is the value with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator>** (const float64 & *value*)
compare operator, compares for “bigger than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

`CmpFunc`

Parameters

- *value* - is the value with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator<=** (const float64 & *value*)
compare operator, compares for “lower or equal than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

`CmpFunc`

Parameters

- *value* - is the value with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator>=** (const float64 & *value*)
compare operator, compares for “bigger or equal than”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

`CmpFunc`

Parameters

- *value* - is the value with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator==** (const float64 & *value*)
compare operator, compares for “equal to”

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `value` - is the value with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator!=** (const float64 & *value*)
compare operator, compares for "unequal to"

Return

compare matrix of type uint8, which contains 0 or 1, depending on the result of the element-wise comparison

See

CmpFunc

Parameters

- `value` - is the value with which this data object should element-wisely be compared

Exceptions

- `cv::Exception` - if both data objects doesn't have the same size or type

DataObject **operator<<** (const unsigned int *shiftbit*)
high-level operator, which shifts the elements of this data objects by a given number of bits to the left and returns the new data object

Return

new data object with shifted values

See

operator <<=, ShiftLFunc

Parameters

- `shiftbit` - defines the number of bits to shift

DataObject & **operator<<=** (const unsigned int *shiftbit*)
high-level operator, which shifts the elements of this data objects by a given number of bits to the left

Return

reference to this data object

See

ShiftLFunc

Parameters

- `shiftbit` - defines the number of bits to shift

DataObject **operator>>** (const unsigned int *shiftbit*)
high-level operator, which shifts the elements of this data objects by a given number of bits to the right and returns the new data object

Return

new data object with shifted values

See

operator >>=, ShiftRFunc

Parameters

- `shiftbit` - defines the number of bits to shift

DataObject & **operator>>=** (const unsigned int *shiftbit*)

high-level operator, which shifts the elements of this data objects by a given number of bits to the right

Return

reference to this data object

See

ShiftRFunc

Parameters

- `shiftbit` - defines the number of bits to shift

DataObject **operator&** (const *DataObject* & *rhs*)

high-level operator, which executes the element-wise operation “bitwise and” between this data object and a given data object

the result is returned as a newly allocated data object.

Return

new data object, where the result of the operation is stored

See

operator &=, BitAndFunc

Parameters

- `&rhs` - is the matrix which is used for the operator

Exceptions

- `cv::Exception` - if data type is not supported or both data objects differs either in their size or data type

DataObject & **operator&=** (const *DataObject* & *rhs*)

high-level operator, which executes the element-wise operation “bitwise and” between this data object and a given data object

Return

reference to this data object, where the result of the operation is stored

See

BitAndFunc

Parameters

- `&rhs` - is the matrix which is used for the operator

Exceptions

- `cv::Exception` - if data type is not supported or both data objects differs either in their size or data type

DataObject **operator|** (const *DataObject* & *rhs*)

high-level operator, which executes the element-wise operation “bitwise or” between this data object and a given data object

the result is returned as a newly allocated data object.

Return

new data object, where the result of the operation is stored

See

operator `|=`, `BitOrFunc`

Parameters

- `&rhs` - is the matrix which is used for the operator

Exceptions

- `cv::Exception` - if data type is not supported or both data objects differs either in their size or data type

DataObject & **operator**`|=` (const *DataObject* & *rhs*)

high-level operator, which executes the element-wise operation “bitwise or” between this data object and a given data object

Return

reference to this data object, where the result of the operation is stored

See

`BitOrFunc`

Parameters

- `&rhs` - is the matrix which is used for the operator

Exceptions

- `cv::Exception` - if data type is not supported or both data objects differs either in their size or data type

DataObject **operator**`^` (const *DataObject* & *rhs*)

high-level operator, which executes the element-wise operation “bitwise or” between this data object and a given data object

the result is returned as a newly allocated data object.

Return

new data object, where the result of the operation is stored

See

operator `^=`, `BitXorFunc`

Parameters

- `&rhs` - is the matrix which is used for the operator

Exceptions

- `cv::Exception` - if data type is not supported or both data objects differs either in their size or data type

DataObject & **operator**`^=` (const *DataObject* & *rhs*)

high-level operator, which executes the element-wise operation “bitwise xor” between this data object and a given data object

Return

reference to this data object, where the result of the operation is stored

See

`BitXorFunc`

Parameters

- `&rhs` - is the matrix which is used for the operator

Exceptions

- `cv::Exception` - if data type is not supported or both data objects differs either in their size or data type

RetVal **zeros** (const int *type*)

allocates a zero-value matrix of size 1x1 with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *type* - is the desired type-number

RetVal **zeros** (const int *size*, const int *type*)

allocates a zero-value matrix of size 1 x *size* with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *size* - is the desired length of the vector
- *type* - is the desired type-number

RetVal **zeros** (const int *sizeY*, const int *sizeX*, const int *type*)

allocates a zero-value matrix of size *sizeY* x *sizeX* with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *sizeY* - are the number of rows
- *sizeX* - are the number of columns
- *type* - is the desired type-number

RetVal **zeros** (const int *sizeZ*, const int *sizeY*, const int *sizeX*, const int *type*, const unsigned char *continuous* = 0)

allocates a zero-value, 3D- matrix of size *sizeZ* x *sizeY* x *sizeX* with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- `sizeZ` - are the number of matrix-planes
- `sizeY` - are the number of rows
- `sizeX` - are the number of columns
- `type` - is the desired type-number
- `continuous` - indicates whether the data should be in one continuous block (true) or not (false)

RetVal **zeros** (const unsigned char *dimensions*, const int * *sizes*, const int *type*, const unsigned char *continuous* = 0)
high-level, non-templated base function for allocation of new matrix whose elements are all set to zero

Return

retOk

See

ZerosFunc

Parameters

- *dimensions* - indicates the number of dimensions
- **sizes* - is a vector with the same length than *dimensions*. Every element indicates the size of the specific dimension
- *type* - is the desired data-element-type
- *continuous* - indicates whether the data should be in one continuous block (true) or not (false)

RetVal **ones** (const int *type*)
allocates a one-value matrix of size 1x1 with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *type* - is the desired type-number

RetVal **ones** (const int *size*, const int *type*)
allocates a one-value matrix of size 1 x *size* with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *size* - is the desired length of the vector
- *type* - is the desired type-number

RetVal **ones** (const int *sizeY*, const int *sizeX*, const int *type*)
allocates a one-value matrix of size *sizeY* x *sizeX* with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *sizeY* - are the number of rows
- *sizeX* - are the number of columns
- *type* - is the desired type-number

RetVal ones (const int *sizeZ*, const int *sizeY*, const int *sizeX*, const int *type*, const unsigned char *continuous* = 0)

allocates a one-valued, 3D- matrix of size *sizeZ* x *sizeY* x *sizeX* with the given type

Return

retOk

See

zeros, ZerosFunc

Parameters

- *sizeZ* - are the number of matrix-planes
- *sizeY* - are the number of rows
- *sizeX* - are the number of columns
- *type* - is the desired type-number
- *unsigned - char continuous* indicates wether the data should be in one continuous block (true) or not (false)

RetVal ones (const unsigned char *dimensions*, const int * *sizes*, const int *type*, const unsigned char *continuous* = 0)

high-level, non-templated base function for allocation of new matrix whose elements are all set to one

Return

retOk

See

OnesFunc

Parameters

- *dimensions* - indicates the number of dimensions
- **sizes* - is a vector with the same length than *dimensions*. Every element indicates the size of the specific dimension
- *type* - is the desired data-element-type
- *continuous* - indicates wether the data should be in one continuous block (true) or not (false)

RetVal rand (const int *type*, const bool *randMode* = false)

allocates a random-value matrix of size 1x1 with the given type

this function allocates an random value matrix using `cv::randu` for uniform (*randMode* = false) or gaussian noise (*randMode* = true). In case of an integer type, the uniform noise is from min(inclusiv) to max(inclusiv). For floating point types, the noise is between 0(inclusiv) and 1(exclusiv). In case of

an integer type, the gaussian noise mean value is $(\text{max}+\text{min})/2.0$ and the standard deviation is $(\text{max}-\text{min})/6.0$ to max . For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

Return

retOk

See

[*zeros*](#), [ZerosFunc](#)

Parameters

- `type` - is the desired type-number
- `randMode` - switch mode between uniform distributed(false) and normal distributed noise(true)

RetVal rand (const int *size*, const int *type*, const bool *randMode* = false)

allocates a random-value matrix of size 1 x *size* with the given type

this function allocates an random value matrix using `cv::randu` for uniform (`randMode` = false) or gaussian noise (`randMode` = true). In case of an integer type, the uniform noise is from `min(inclusiv)` to `max(inclusiv)`. For floating point types, the noise is between 0(`inclusiv`) and 1(`exclusiv`). In case of an integer type, the gaussian noise mean value is $(\text{max}+\text{min})/2.0$ and the standard deviation is $(\text{max}-\text{min})/6.0$ to max . For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

Return

retOk

See

[*zeros*](#), [ZerosFunc](#)

Parameters

- `size` - is the desired length of the vector
- `type` - is the desired type-number
- `randMode` - switch mode between uniform distributed(false) and normal distributed noise(true)

RetVal rand (const int *sizeY*, const int *sizeX*, const int *type*, const bool *randMode* = false)

allocates a random-value matrix of size *sizeY* x *sizeX* with the given type

this function allocates an random value matrix using `cv::randu` for uniform (`randMode` = false) or gaussian noise (`randMode` = true). In case of an integer type, the uniform noise is from `min(inclusiv)` to `max(inclusiv)`. For floating point types, the noise is between 0(`inclusiv`) and 1(`exclusiv`). In case of an integer type, the gaussian noise mean value is $(\text{max}+\text{min})/2.0$ and the standard deviation is $(\text{max}-\text{min})/6.0$ to max . For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

Return

retOk

See

[*zeros*](#), [ZerosFunc](#)

Parameters

- `sizeY` - are the number of rows
- `sizeX` - are the number of columns
- `type` - is the desired type-number

- `randMode` - switch mode between uniform distributed(false) and normal distributed noise(true)

RetVal **rand** (const int *sizeZ*, const int *sizeY*, const int *sizeX*, const int *type*, const bool *randMode*, const unsigned char *continuous* = 0)
allocates a random-valued, 3D- matrix of size *sizeZ* x *sizeY* x *sizeX* with the given type

this function allocates an random value matrix using `cv::randu` for uniform (`randMode` = false) or gaussian noise (`randMode` = true). In case of an integer type, the uniform noise is from min(inclusiv) to max(inclusiv). For floating point types, the noise is between 0(inclusiv) and 1(exclusiv). In case of an integer type, the gaussian noise mean value is $(\text{max}+\text{min})/2.0$ and the standard deviation is $(\text{max}-\text{min})/6.0$ to max. For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

Return

`retOk`

See

[*zeros*](#), [*ZerosFunc*](#)

Parameters

- *sizeZ* - are the number of matrix-planes
- *sizeY* - are the number of rows
- *sizeX* - are the number of columns
- *type* - is the desired type-number
- *randMode* - switch mode between uniform distributed(false) and normal distributed noise(true)
- *unsigned* - char *continuous* indicates wether the data should be in one continuous block (true) or not (false)

RetVal **rand** (const unsigned char *dimensions*, const int * *sizes*, const int *type*, const bool *randMode*, const unsigned char *continuous* = 0)

high-level, non-templated base function for allocation of new matrix whose elements are all set to one

this function allocates an random value matrix using `cv::randu` for uniform (`randMode` = false) or gaussian noise (`randMode` = true). In case of an integer type, the uniform noise is from min(inclusiv) to max(exclusive). For floating point types, the noise is between 0(inclusiv) and 1(exclusiv). In case of an integer type, the gaussian noise mean value is $(\text{max}+\text{min})/2.0$ and the standard deviation is $(\text{max}-\text{min})/6.0$ to max. For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

Return

`retOk`

See

[*OnesFunc*](#)

Parameters

- *dimensions* - indicates the number of dimensions
- **sizes* - is a vector with the same length than *dimensions*. Every element indicates the size of the specific dimension
- *type* - is the desired data-element-type
- *randMode* - switch mode between uniform distributed(false) and normal distributed noise(true)
- *continuous* - indicates wether the data should be in one continuous block (true) or not (false)

RetVal **eye** (const int *type*)

sets the matrix of this data object to a two-dimensional eye-matrix of size 1, hence [1]

Return

retOk

See

ones

Parameters

- *type* - is the desired element data-type

RetVal **eye** (const int *size*, const int *type*)

sets the matrix of this data object to a two-dimensional eye-matrix of given size

At first, a preexisting matrix is freed, before creating the eye-matrix

Return

retOk

See

freeData, create, EyeFunc

Parameters

- *size* - is the desired size of the squared eye-matrix
- *type* - is the desired element data-type

RetVal **conj** ()

converts every element of the data object to its conjugate complex value

Return

retOk

See

ConjFunc

Exceptions

- `cv::Exception` - if data type is not complex.

DataObject **adj** () const

converts every element of the data object to its adjungate value

The adjungate is the transposed matrix, where each element is complex conjugated.

Return

retOk

See

conj

Exceptions

- `cv::Exception` - if data type is not complex.

DataObject **trans** () const

transposes this data object

simply toggles the transpose flag

Return

reference to this data object

DataObject **mul** (const *DataObject* & *mat2*, const double *scale* = 1.0) const

high-level method which does a element-wise multiplication of elements in this matrix with elements in the second matrix.

The result is returned as new data object with the same type and size than this object. The axis scale, offset, description and unit values are copied from this object. Tags are copied from this object, too. Optionally the multiplication can be scaled by a scaling factor, which is set to one by default.

Return

result matrix

See

DivFunc

Parameters

- *&mat2* - is the second source matrix
- *scale* - is the scaling factor (default: 1.0)

DataObject **div** (const *DataObject* & *mat2*, const double *scale* = 1.0) const

high-level method which does a element-wise division of elements in this matrix by elements in second source matrix.

The result is returned as new data object with the same type and size than this object. The axis scale, offset, description and unit values are copied from this object. Tags are copied from this object, too.

Return

result matrix

See

DivFunc

Parameters

- *&mat2* - is the second source matrix
- *scale* - is the scaling factor (default: 1.0)

int **elemSize** () const

returns number of bytes required by each value in the array.

number of bytes that are required by each value inside of the data object array (e.g. 1 for uint8, 2 for int16...)

Return

the size of each array element in bytes.

template <typename *_Tp*>

const *_Tp* & **at** (const unsigned int *y*, const unsigned int *x*) const
addressing method for two-dimensional data object.

Return

const reference to specific element

Parameters

- *y* - is the zero-based row-index to the element which is requested (considering any ROI)
- *x* - is the zero-based column-index to the element which is requested (considering any ROI)

template <typename *_Tp*>

_Tp & **at** (const unsigned int *y*, const unsigned int *x*)
addressing method for two-dimensional data object.

Return

reference to specific element

Parameters

- *y* - is the zero-based row-index to the element which is requested (considering any ROI)
- *x* - is the zero-based column-index to the element which is requested (considering any ROI)

```
template <typename _Tp>
const _Tp & at (const unsigned int z, const unsigned int y, const unsigned int x) const
    addressing method for three-dimensional data object.
```

Return

const reference to specific element

Parameters

- *z* - is the zero-based z-index to the element which is requested (considering any ROI)
- *y* - is the zero-based row-index to the element which is requested (considering any ROI)
- *x* - is the zero-based column-index to the element which is requested (considering any ROI)

```
template <typename _Tp>
_Tp & at (const unsigned int z, const unsigned int y, const unsigned int x)
    addressing method for three-dimensional data object.
```

Return

reference to specific element

Parameters

- *z* - is the zero-based z-index to the element which is requested (considering any ROI)
- *y* - is the zero-based row-index to the element which is requested (considering any ROI)
- *x* - is the zero-based column-index to the element which is requested (considering any ROI)

```
template <typename _Tp>
const _Tp & at (const unsigned int * idx) const
    addressing method for n-dimensional data object.
```

Remark

The *idx* vector must indicate the indices in “virtual”-order (user-friendly order)

Return

const reference to specific element

Parameters

- **idx* - is vector whose size is equal to the data object’s dimensions. Each entry indicates the zero-based index of its specific dimension considering any ROI

```
template <typename _Tp>
_Tp & at (const unsigned int * idx)
    addressing method for n-dimensional data object.
```

Remark

The *idx* vector must indicate the indices in “virtual”-order (user-friendly order)

Return

reference to specific element

Parameters

- `*idx` - is vector whose size is equal to the data object's dimensions. Each entry indicates the zero-based index of its specific dimension considering any ROI

DataObject **at** (const ito::Range & *rowRange*, const ito::Range & *colRange*) const

addressing method for two-dimensional data object with two given range-values. returns shallow copy of addressed regions.

addressing method for two-dimensional data object with two given range-values. returns shallow copy of addressed regions

Return

new data object which is a shallow copy of this data object and whose ROI is set to the given row- and col-ranges

Parameters

- *rowRange* - is the desired rowRange which should be in the new ROI (considers any existing ROI, too)
- *colRange* - is the desired colRange which should be in the new ROI (considers any existing ROI, too)

Exceptions

- `cv::Exception` - if number of dimensions is unequal to two.

DataObject **at** (ito::Range * *ranges*) const

addressing method for n-dimensional data object with n given range-values. returns shallow copy of addressed regions

If any of the given ranges exceed the boundaries of its corresponding dimension, the range will be set to the boundaries. ranges will be given in “virtual” order, hence, the transpose-flag is considered by this method.

Return

new data object with shallow copy of this data object and adjusted ROI with respect to the given ranges

See

GetRangeFunc

Parameters

- **ranges* - is vector of desired ranges for each dimension

DataObject **at** (const *DataObject* & *mask*) const

addressing method that returns a 1xM data object of the same type than this object with only values that are marked in the given uint8 mask object

addressing method that returns a Mx1 data object of the same type than this object with only values that are marked in the given uint8 mask object

This method returns a new 1xM data object with the same type than this data object. The M columns are filled with a values of this data object whose corresponding mask value is != 0.

Return

new data object with shallow copy of this data object and adjusted ROI with respect to the given ranges

Parameters

- *mask* - is a uint8 mask data object with the same size than this object. Values != 0 are valid values in the mask.

uchar * **rowPtr** (const int *matNum*, const int *y*)
 returns pointer to the data in the *y*-th row in the 2d-matrix plane *matNum*
 cast this pointer to the data type of the matrix elements (as pointer).

Remark

No further error checking (e.g. boundaries)

Return

data-pointer

const uchar * **rowPtr** (const int *matNum*, const int *y*) const
 returns pointer to the data in the *y*-th row in the 2d-matrix plane *matNum*
 cast this pointer to the data type of the matrix elements (as pointer).

Remark

No further error checking (e.g. boundaries)

Return

data-pointer

DataObject **row** (const int *selRow*) const

low-level, templated method which changes the region of interest of the data object to the selected zero-based row index

Return

retOk high-level method which makes a new header for the specified matrix row and returns it.
 The underlying data of the new matrix is shared with the original matrix.

Return

new data object

See

RowFunc

Parameters

- **dObj* -
- *selRow* - indicates the zero-based row-index (considering any existing ROI)

Parameters

- *selRow* - is the specific zero-based row index

Exceptions

- *cv::Exception* - if dimension is unequal to two.

DataObject **col** (const int *selCol*) const

low-level, templated method which changes the region of interest of the data object to the selected zero-based col index

Return

retOk high-level method which makes a new header for the specified matrix column and returns it. The underlying data of the new matrix is shared with the original matrix.

Return

new data object

See

ColFunc

Parameters

- `*dObj` -
- `unsigned - int selCol` indicates the zero-based col-index (considering any existing ROI)

Parameters

- `selCol` - is the specific zero-based row index

Exceptions

- `cv::Exception` - if dimension is unequal to two.

DataObject **toGray** (const int *destinationType* = `ito::tUInt8`) const
converts a color image (rgba32) to a gray-scale image

usage: `res = static_cast<ito::float32>(sourceDataObject)`

Return

cast data object

See

convertTo, CastFunc

Exceptions

- `cv::Exception` - if cast failed, e.g. if cast not possible or types unknown

DataObject & **adjustROI** (const int *dtop*, const int *dbottom*, const int *dleft*, const int *dright*)
adjust submatrix size and position within the two-dimensional data-object

changes the boundaries of the ROI of a two-dimensional data object by the given incremental values

Remark

the parameters indicates the shift with respect to the virtual order of the matrix, hence, the transpose flag is considered in this method

Return

reference to this data object

See

adjustROI

Parameters

- `dtop` - The shift of the top submatrix boundary upwards (positive value means upwards)
- `dbottom` - The shift of the bottom submatrix boundary downwards (positive value means downwards)
- `dleft` - The shift of the left submatrix boundary to the left (positive value means to the left)
- `dright` - The shift of the right submatrix boundary to the right (positive value means to the right)

Exceptions

- `cv::Exception` - if data object is not two-dimensional

DataObject & **adjustROI** (const unsigned char *dims*, const int * *lims*)
adjust submatrix size and position within the n-dimensional data-object

changes the boundaries of the ROI of a n-dimensional data object by the given incremental values

`dims` is the number of dimensions

Return

reference to this data object

Remark

`lims` indicates the shift with respect to the virtual order of the matrix, hence, the transpose flag is considered in this method

See

[adjustROI](#)

Parameters

- `*lims` - is a integer array whose length is $2 \times \text{dims}$. For every dimension, two adjacent values indicates the shift of the ROI. The first of both values indicates the shift of the ROI towards the first element in the matrix (positive direction). The second value indicates the shift of the ROI towards the last element in the matrix (positive direction).

RetVal **locateROI** (int * *wholeSizes*, int * *offsets*) const

method locates ROI of this data object within its original data block

locates the boundaries of the ROI of a n-dimensional data object and returns the original size and the distances to the physical borders

long description

Return

retOk

Parameters

- `*wholeSizes` - is an allocated array of size `m_dims`, which is filled with the original matrix-sizes (considering the transpose-flag, hence, the output is in user-friendly form)
- `*offsets` - is dimension-wise offset in order to get from the original first element of the matrix to the subpart within the region of interest, array must be pre-allocated, too.

RetVal **locateROI** (int * *lims*) const

method get ROI of this data object within its original data block

locates the boundaries of the ROI of a n-dimensional data object the distances to the physical borders

`dims` is the number of dimensions

Return

retOk

Parameters

- `*lims` - is a integer array whose length is $2 \times \text{dims}$. For every dimension, two adjacent values indicates the shift of the ROI. The first of both values indicates the shift of the ROI towards the first element in the matrix (positive direction). The second value indicates the shift of the ROI towards the last element in the matrix (positive direction).

template <typename *_Tp*>

RetVal **copyFromData2D** (const *_Tp* * *src*, const int *sizeX*, const int *sizeY*)

copies the externally given source data inside this data object

This method obtains an externally given source array that must have the same element type than this data object. Its dimension is given by `sizeX` and `sizeY` and must correspond to the x-size and y-size of this data object. It is allowed that this data object is a shallow copy with a possible region of interest of another (bigger) object.

Then, the given array is copied inside of the values of the data object. The external array must have a row-wise data arrangement (c-style), hence, one row follows after the other one.

Return

RetVal error if sizeX or sizeY does not fit to the size of the data object or if the type of the given array does not fit to the type of the data object

Parameters

- `_Tp*` - src is the source array. The type of the array is analyzed at compile time (`_Tp` is the placeholder for this type as template parameter)
- `sizeX` - is the width of the array and must fit to the plane width of the data object
- `sizeY` - is the height of the array and must fit to the plane height of the data object

template <typename `_Tp`>

RetVal **copyFromData2D** (const `_Tp` * *src*, const int *sizeX*, const int *sizeY*, const int *x0*, const int *y0*, const int *width*, const int *height*)

copies the externally given source data inside this data object

This method obtains an externally given source array that must have the same element type than this data object. Its dimension is given by `sizeX` and `sizeY` and must correspond to the x-size and y-size of this data object. It is allowed that this data object is a shallow copy with a possible region of interest of another (bigger) object.

Then, the given array is copied inside of the values of the data object. The external array must have a row-wise data arrangement (c-style), hence, one row follows after the other one.

In this method, it is allowed that the original width and height of the given data is different than the plane size of this data object. Then only a subregion of the external data is copied, indicated by the `x0` and `y0` indices of the first value and its width and height (`sizeX` and `sizeY` are the original size of the given data). width and height must correspond to the plane size of the data object.

Return

RetVal error if sizeX or sizeY does not fit to the size of the data object or if the type of the given array does not fit to the type of the data object

Parameters

- `_Tp*` - src is the source array. The type of the array is analyzed at compile time (`_Tp` is the placeholder for this type as template parameter)
- `sizeX` - is the width of the array.
- `sizeY` - is the height of the array.
- `x0` - is the x-index of the first value of the source data that is copied.
- `y0` - is the y-index of the first value of the source data that is copied.
- `width` - is the width of the sub-region of the source data that should be copied (must fit to the width of the data object)
- `height` - is the height of the sub-region of the source data that should be copied (must fit to the height of the data object)

template <typename `T2`>

operator `T2` ()

cast operator for data object

cast operator, tries to cast this data object to another element type

usage: `res = static_cast<ito::float32>(sourceDataObject)`

Return

cast data object

See*convertTo*, CastFunc**Exceptions**

- `cv::Exception` - if cast failed, e.g. if cast not possible or types unknown

template <typename _Tp>

RetVal **linspace** (const _Tp start, const _Tp end, const _Tp inc, const int transposed)

equivalent to matlab linspace functino

Return**See**

MakeContinuousFunc

Parameters

- `&dObj` - is the source data object

8.3.4 Parameter-Container class of itom

Introduction

The base idea behind a container-class for parameters of varying types is to pass these parameters to methods without the need of extensive templating. Of course, both **Python** and **Qt** provide such classes, namely **PyObject** and **QVariant**. Nevertheless, **itom** provides an own, low-level container class, which is not dependent on any 3rd party library. This container only provides support for some specific types, that are widely used within **itom**. This parameter container is mainly used for the whole communication process between all types of plugins and **itom**. Examples for their use are:

- Parameters of each plugin, which can be read by *getParam* or set by *setParam*. Additionally these parameters of the plugin can be obtained by the python command `getParamList()`.
- Mandatory and optional parameters for the constructor of plugin instances.
- Parameter transfer between configuration dialog, docking widget toolbox and plugin itself.

The whole implementation of the container-class is found in the files *sharedStructures.h* and *sharedStructures.cpp*, both lying in the folder *common*, where you can also find the file *addInInterface.h*.

The possible types, covered by this container are:

- **char** (Type 1)
- **integer** (Type 1)
- **double** (Type 1)
- **char-array** (Type 1)
- **integer-array** (Type 1)
- **double-array** (Type 1)
- **string** (zero-terminated, Type 1)
- **DataObject** (pointer-only, Type 2)
- **PointCloud** (pointer-only, Type 2)
- **PolygonMesh** (pointer-only, Type 2)
- **HWRef** (pointer-only, points to an instance of *AddInBase*, Type 2)

Note: *Type 1* means, that these types are internally copied when calling a constructor, copy constructor or assignment operator of the parameter, where for parameters of *Type 2* only the pointer to the value itself is internally stored in the parameter, hence, only this pointer is copied at the above mentioned methods. The reason is, that a quick passing of the parameter is provided, on the other side, parameters of *Type 2* need some further attention concerning thread-safety and/or creation and deletion responsibility.

***ParamBase* and *Param* and the *Meta*-classes**

There are different classes, defined in *sharedStructures.h* which can be used for the parameter container:

- class **ParamBase** represents a pure multi-type container, which only contains the name of the parameter and its value. This is the quickest parameter container implementation and should always be used, if only the parameter should be passed and no further information are needed. In any other cases use an implementation of the derived class *Param*.
- class **Param** is derived from *ParamBase* and additionally contains a description string for the value (optional) and some further meta-information (optional) in form of an internal pointer, that points to an instance of class *MetaParam* or one of its derived classes.
- class **MetaParam** is the base class for all parameter type-dependent meta information classes, like *IntMeta*, *DoubleMeta*, *StringMeta*... The idea is to add some restrictions about value ranges, allowed values... to parameters if this is needed or available. Please consider, that the meta information class is not internally checked in the parameter classes, hence, you can assign every value you want to. However the programmer, that is using a parameter, can access the *MetaParam*-instance and program its own validator for the parameter or use one of the pre-defined methods.
- class **CharMeta**, **IntMeta**, **DoubleMeta** are meta information classes derived from *MetaParam* which contain a minimum and maximum value for the parameter. Parameters of array-types may also contain an instance of one of these classes in order to describe the allowed range of every element of the array.
- class **StringMeta** provides further information for the parameter of type *String* such that you restrict the string to certain values, which also can be evaluated in the sens of a regular expression or wildcard expression.
- class **HWMeta** provides restrictive information for a parameter of type *HWRef*

For more information about the meta information classes see [Parameters - Meta Information](#).

Usage and differences of the classes *ParamBase* and *Param*

Variables of class **Param** are used whenever you explicitly want to add further information to your parameter. Examples might be:

- Vectors of mandatory and optional parameters, used as template for creating an instance of a plugin.
- Vectors of mandatory and optional parameters, used as template for creating a widget defined in a plugin of type *AddInAlgo*.
- Vectors of mandatory, optional and out parameters, used as template for calling a filter.
- Plugin-internal parameters, stored in the Map *m_params*.
- *getParam*-method of plugins, which usually return one specific value of map *m_params*. Here an instance of class *Param* is returned and not *ParamBase* such that advanced information about the value can be presented.
- Vector of out-parameters of filters.

Variables of class *ParamBase* are used when you only need to transfer the parameter itself:

- Method *setParam* of plugins. The validation of the given value is done with respect to its corresponding value in map *m_params*.
- Vector of mandatory and optional parameters used for calling the constructor (method *init*) of plugins.

- Vector of mandatory, optional and out parameters used for calling filters in plugins.
- Vector of mandatory and optional parameters used for calling widgets, defined in plugins of type *AddInAlgo*.

In the case of the described mandatory and optional parameter vectors, **itom** is requesting the template version (class *Param*) from the plugin and has enough information, in order to check the user input (done in GUI or by python) with respect to the template. Finally a vector of type *ParamBase* is created, where all the default values, given by the templates, are overwritten by the user input. Then the filter, plugin constructor or widget constructor is called with the version of *ParamBase*.

Types and flags

The type as well as additional flags of each parameter is defined by an OR combination of values, contained in the enumeration **ito::ParamBase::Type**. The last 16 bit (bit 1-16) of this enumeration are reserved for type information, the first 16 bits may contain flags.

They can be separated using an AND-operation with the masks **ito::paramFlagMask** or **ito::paramTypeMask**:

```
int typesAndFlags
int types = typesAndFlags & ito::paramTypeMask
int flags = typesAndFlags & ito::paramFlagMask
```

The following (high-level) types are available:

```
enum Type {
    ...
    //type (bit 1-16)
    Pointer      = 0x000001, //do not use directly
    Char         = 0x000002, //Character-Parameter
    Int          = 0x000004, //Integer-Parameter
    Double       = 0x000008, //Double-Parameter
    String       = 0x000020 | Pointer, //zero-terminated String-Parameter
    HWRef        = 0x000040 | Pointer | NoAutosave, //pointer to plugin-instance
    DObjPtr      = 0x000010 | Pointer | NoAutosave, //pointer to dataObject
    CharArray    = Char      | Pointer, //array of characters
    IntArray     = Int       | Pointer, //array of integers
    DoubleArray  = Double    | Pointer, //array of doubles
    PointCloudPtr = 0x000080 | Pointer | NoAutosave, //pointer to point cloud
    PointPtr     = 0x000100 | Pointer | NoAutosave, //pointer to point
    PolygonMeshPtr = 0x000200 | Pointer | NoAutosave //pointer to polygon mesh
    ...
};
```

Note: All pointer-based types have the **NoAutosave**-flag, since a pointer can not be saved to harddrive. The arrays of **CharArray**, **IntArray** and **DoubleArray** are internally copied (e.g. in a copy-constructor), therefore only use them for smaller arrays and not for matrices with millions of entries. This might be an inefficient structure though.

The following flags are implemented in the **Type**-enumeration:

```
enum Type {
    NoAutosave,
    Readonly,
    In,
    Out
    ...
};
```

The behaviour of the **NoAutosave**-flag can be read in see [Automatic loading and saving of plugin parameters](#). The **readonly**-flag marks this parameter to be readonly. Please consider, that this flag is not evaluated in the classes **Param** or **ParamBase**, but the programmer has access to this flag must implement the necessary behaviour. The flags **In** and **Out** or their combination are important for the declaration of the default parameters for plugins or

filter-calls. If none of them is set, the flag **In** is automatically set. **In** indicates, that the parameter is handled like an input-variable only, hence, the filter or plugin's init method will not change the value of this parameter. A variable of type **InOut** passes a value and the value might be changed within a filter call. This is a suitable form to pass a dataObject whose content and size might be changed by the filter. Parameters with flag **Out** only are only accepted in the parameter vector which is the default for the output-variables of a filter... It is only allowed to mark parameter of type **Char**, **Int**, **Double**, **String**, **CharArray**, **IntArray** or **DoubleArray** as **Out**- parameters.

Class *ParamBase*

The class `ParamBase` consists of the following main elements or member variables, which however are only accessible by corresponding getter- or setter-methods:

m_type

This variable contains an OR combination of the data type, covered by the parameter container as well as some additional flags (read-only, auto-save). Read the section *Types and flags* for more information about the type.

The type part of this member is obtained by `getType()`, the flags can be obtained by `getFlags()`.

ito::ByteArray **m_name**

This member contains the name of the parameter. This name is for example used for accessing the parameter in the python's `setParam` or `getParam` method and usually you can also use this name as keyword in a python argument list of appropriate method calls.

Access the name of a parameter by using `getName()`. This returns the zero-terminated name string as char-pointer.

values

There are three further member variables which are used in order to store the variable content of the parameter container. Reading and writing these values is only done by the constructor or the methods `getVal<_Tp>` and `setVal<_Tp>`.

Typical creations for parameters of class **ParamBase** are:

```
//empty parameter (name: NULL, type: 0)
ParamBase p1;

//creating an integer-parameter, flag: In, value: 2
ParamBase p2("IntParam", ito::ParamBase::Int | ito::ParamBase::In, 2);

//creating a double-parameter, flag: Readonly, value: -4.0
ParamBase p3("Name", ito::ParamBase::Double | ito::ParamBase::Readonly, -4.0);

//creating a string-parameter
ParamBase p4("Name", ito::ParamBase::String, "default text");

//creating an integer-array parameter
int size = 5;
int* a = new int[size];
//.. fill a with valid values
ParamBase p5("Array", ito::ParamBase::IntArray, size, a);
//a is copied, therefore delete it now
delete[] a;
a = NULL;

//passing a dataObject pointer as parameter
ito::DataObject *dObj = new ito::DataObject(...);
ParamBase p6("param", ito::ParamBase::DObjPtr, dObj);
//be careful: p6 only holds a pointer to dObj, therefore you can only delete it
// if p6 does not access it any more.

//passing a pointer to another actuator- or dataIO-instance to a parameter
ito::AddInActuator *aia = ...;
```

```
ParamBase p7("motor", ito::ParamBase::HWRref, aia);
//like with the dataObject. Be careful and make sure, that the pointer 'aia' remains
//accessible during the lifetime of p7.
```

The parameter **p1** has no suitable type or value right now. However, you can assign another parameter to **p1** by using the assignment operator:

```
p1 = ParamBase("newVal", ito::ParamBase::Char, 128)
```

If you have an array-parameter, you can access one single index of this array, which is then returned as new instance of **ParamBase**. If the index is out of range or the parameter is no array-type, an empty instance of **ParamBase** is returned:

```
//use p5 from the example above
ParamBase p5_0 = p5[0] //is a valid parameter of type Int
ParamBase p5_5 = p5[5] //error. empty ParamBase since index exceeded the maximum size.
```

Reading values from the parameter is done by the method **getVal**. This method must be called with a template parameter, that corresponds to the original data type, which is covered by the parameter.

```
//This example is based on the constructed params above
int p2_val = p2.getVal<int>();

double p3_val = p3.getVal<double>();

//the following examples return the internal pointer to the string or arrays.
//This pointer is no copy, therefore you are not allowed to delete the pointer.
char* p4_val = p4.getVal<char*>();

int* p5_val = p5.getVal<int*>();
//you can access the elements of p5 by
int temp = p5_val[0];
temp = p5_val[4];
//p5_val[5] is not allowed, since it exceeds the number of elements of this array

//in order to get the number of values in the parameter, use the following
//implementation
int length = 0;
p5_val = p5.getVal<int*>(length);
//now length is equal to 5.

//pointer-parameters are obtained by using the right template
//parameter of the getVal method. The internal pointer of the
//parameter is then casted to the template type.
ito::DataObject *dObj = p6.getVal<ito::DataObject*>();

ito::AddInActuator *aia = p7.getVal<ito::AddInActuator*>();

//If you are sure that the parameter contains at least any plugin, however you have no idea
//whether it is an acutator or an instance of dataIO, you could at first get the
//base instance to ito::AddInBase and then try to safely cast it to your requested type:
ito::AddInBase *aib = p7.getVal<ito::AddInBase*>();
ito::AddInActuator *aia = qobject_cast<ito::AddInActuator*>(aib);
//aia is NULL, if the cast failed.
```

If the given template parameter does not fit to the corresponding parameter type, the value of the parameter will be casted to the given template type. If this is not possible an exception is raised. The exception is of type **std::logic_error**.

Settings values to the parameter can be analogously done by the method **setVal**. This also is a template based method. The following code snippets show examples how to change values of the previously constructed parameters **p1** to **p7**:

```
ito::RetVal retVal;
retVal += p2.setVal<int>(5); //retVal remains retOk
retVal += p3.setVal<double>(-3.7); //retVal remains retOk

//p4 is a string-type. New values assigned to p4 (here: "new value")
// are internally copied.
retVal += p4.setVal<char*>("new value"); //retVal remains retOk

//for array-types, you can only assign the whole array and not change any
//elements. For changing values, use getVal<_Tp>(...) in order to obtain the pointer and
//change the values directly.
int values[] = {1,2,3,4,5};
retVal += p5.setVal<int*>(values,5); //always provide the length of the array
//again, the setVal-method above internally copies the array and you can destroy the
//source.
int length = p5.getLen(); //length of array

ito::DataObject dObj;
retVal += p6.setVal<void*>((void*)&dObj);
//again remember, that p6 requires, that dObj remains accessible during the lifetime of p6.
```

Note: The method **setVal** will never change the assigned type of the parameter. If the new value can not be converted into the internal type of the parameter, **setVal** will return with a **RetVal**, that contains errors.

If you only want to copy the content of one parameter of type **ParamBase** to your parameter, then you can use the method **copyValueFrom**, which requires another instance of **ParamBase** or **Param** (since this is derived from **ParamBase**). The method returns an error if the parameters are not compatible:

For a full reference to all member function of class **ParamBase**, see [ParamBase - Reference](#).

Class **Param**

The class **Param** is derived from `:cpp:class'ParamBase'`. Therefore it has all features of **ParamBase** including two additional member variables:

ParamMeta ***m_pMeta**

This is a pointer to a struct containing type-dependent meta information about this plugin. This pointer may also be NULL, if no meta information is provided. The meta-information struct is always owned by the parameter and deeply copied when calling for instance a copy constructor. For more information see [Parameters - Meta Information](#).

Access to the meta information struct is given by

```
Param p;
ParamMeta* meta = p.getMeta();
//in this case meta is NULL, since no meta information has been set to 'p'.

//now we create a integer-variable with a min and max value
Param p2("var1",ParamBase::Int,2);
IntMeta meta2(-2,2);

//now we set the meta information of p2 to meta2. Since the ownership of meta2 should not
//be taken by p2, the second argument is false. Then, p2 makes a copy of meta2.
p2.setMeta(&meta2,false);

meta = p.getMeta();
//meta is now a pointer to a structure of type ParamMeta. It can be casted to IntMeta.
```

ito::ByteArray **m_Info**

This is the description string of the parameter. If no description is indicated, this pointer is NULL, else it is a zero-terminated string, which is also copied, when the parameter is called using a copy constructor or assigned to another parameter.

The description can be obtained by

```
Param p("name", ParamBase::String, "content", "information")
const char* descr = p.getInfo(); //descr is 'information'
```

Note: Do not delete the char-pointer returned by `getInfo`, since this is only a reference to the internal description string of the parameter.

The description string is changed by

```
p.setInfo("new information")
```

The full reference of class `Param` is available in *Param - Reference*.

In the following, examples about how to create parameters and meta information of different types are shown:

- **Integer-Type (Type: Int)**

This is one fixed-point number in the integer-range.

```
//integer value between 0 and 10, default: 5
ito::Param param("intNumber", ito::ParamBase::Int, 0, 10, 5, "description");
//this default constructor automatically creates an internal meta-information struct
//of class IntMeta.

// or (here param becomes owner of IntMeta-instance)
ito::Param param("intNumber", ParamBase::Int, 5, new IntMeta(0,10), "description");

// or (integer-variable without meta information)
ito::Param param("intNumber", ParamBase::Int, 5, NULL, "description");
param.setMeta(new IntMeta(0,10), true); //take ownership of IntMeta-instance

int value = param.getVal<int>(); //returns 5
ito::RetVal retValue = param.setVal<int>(6); //returns ito::retOk
bool numeric = param.isNumeric() //returns true, since integer is a numeric

// accessing the min-max-value is obtained by getting the IntMeta-struct
IntMeta *meta = dynamic_cast<IntMeta*>(param.getMeta());
if(meta) //meta is only valid, if it has been assigned.
{
    int min = meta->getMin() //returns 0
    int max = meta->getMax() //returns 10
}
int len = param.getLen() //returns 1
```

- **Double-Type (Type: Double)**

This is one floating-point number in the double-range.

```
//integer value between 0.0 and 10.0, default: 5.0
ito::Param param("doubleNumber", ito::ParamBase::Double, 0.0, 10.0, 5.0, "description");
//this default constructor automatically creates an internal meta-information struct
//of class DoubleMeta.

// or (here param becomes owner of DoubleMeta-instance)
ito::Param param("doubleNumber", ParamBase::Double, 5.0, DoubleMeta::all(), "description");
// the command DoubleMeta::all() creates a new instance of DoubleMeta, where the boundaries
// are the minimum and maximum possible value of the double-range.

// or (double-variable without meta information)
ito::Param param("doubleNumber", ParamBase::Double, 5.0, NULL, "description");
param.setMeta(new DoubleMeta(0.0,10.0), true); //take ownership of DoubleMeta-instance

double value = param.getVal<double>(); //returns 5.0
ito::RetVal retValue = param.setVal<double>(6.0); //returns ito::retOk
```

```
bool numeric = param.isNumeric() //returns true, since integer is a number

// accessing the min-max-value is obtained by getting the DoubleMeta-struct
DoubleMeta *meta = dynamic_cast<DoubleMeta*>(param.getMeta());
if(meta) //meta is only valid, if it has been assigned.
{
    double min = meta->getMin()
    double max = meta->getMax()
}
int len = param.getLen() //returns 1
```

- **String-Type (Type: String)**

This is one zero-terminated String.

```
ito::Param param("string", ito::ParamBase::String, "", "description");

//if you want to provide a string-meta information, you must do it in the following separate
ito::StringMeta *meta = new ito::StringMeta(ito::StringMeta::String);
meta->addItem("yes");
meta->addItem("no");
//the meta information indicates, that only the exact matches of both values "yes" or "no"
//might be accepted by this parameter.
param.setMeta(meta, true); //takes ownership of meta

char* value = param.getVal<char*>(); //returns the pointer to the internally saved string
ito::RetVal retValue = param.setVal<char*>("test"); //should return ito::retOk, String is
bool numeric = param.isNumeric() //returns false
int len = param.getLen() //0 if empty string, else length of string
```

Note: Please do not delete the pointer to the internally saved string, obtained by getVal<char*>()!

- **Array of char values (Type: CharArray)**

This is an array of character values. Consider that you should use the constructor where you can give the length of the array, else an error is returned.

```
char ptr = [0,56,127,-10,-20];
ito::Param param("array", ito::ParamBase::CharArray, 5, &ptr, "description");
//you can add a meta-information struct of class CharMeta to that char-array (if desired)

char* value = param.getVal<char*>(); //returns the pointer to the first element of the array
ito::RetVal retValue = param.setVal<char*>(ptr,5); //should return ito::retOk
bool numeric = param.isNumeric() //returns false (even it is an array of numeric values)
int len = param.getLen() //5
ito::Param param0 = param[0]; //returns a char-parameter with value 0
```

- **Array of integer values (Type: IntArray)**

This is an array of integer values. Consider that you should use the constructor where you can give the length of the array, else an error is returned.

```
int ptr = [1,2,3,4,5];
ito::Param param("array", ito::ParamBase::IntArray, 5, &ptr, "description");
//you can add a meta-information struct of class IntMeta to that integer array (if desired)

int* value = param.getVal<int*>(); //returns the pointer to the first element of the array
ito::RetVal retValue = param.setVal<int*>(ptr,5); //should return ito::retOk
bool numeric = param.isNumeric() //returns false (even it is an array of numeric values)
int len = param.getLen() //5
ito::ParamBase param2 = param[1] //returns integer-parameter (casted to ParamBase)
```

- **Array of integer values (Type: DoubleArray)**

This is an array of double values. Consider that you should use the constructor where you can give the length of the array, else an error is returned.

```
double ptr = [1.2,2.3,3.4,4.1,5.2];
ito::Param param("array", ito::ParamBase::DoubleArray, 5, &ptr, "description");

double* value = param.getVal<double*>(); //returns the pointer to the first element of
ito::RetVal retValue = param.setVal<double*>(ptr,5); //should return ito::retOk
bool numeric = param.isNumeric() //returns false (even it is an array of numeric v
int len = param.getLen() //5
```

- **Reference to any initialized instance of dataIO or actuator (Type: HWRef)**

This is the reference to any other initialized instance of dataIO or actuator. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *NoAutosave* is always set for that type. If such a parameter is passed to the *init*-method of a plugin, the reference of the passed plugin is automatically increased (marks the plugin as being used by the new plugin) and vice-versa decremented when the new plugin is closed again.

You can restrict the allowed plugin-references to types which have a minimum amount of bits in the plugin's type bitmask set. Further you can decline a restriction by indicating the exact name of a plugin. Please consider, that this check is passed in form of a class **HWMeta**, however the check is not executed by classes **Param** or **ParamBase**. You have to check this manually.

```
ito::Param param("serialPort", ito::ParamBase::HWRef, NULL, "description");

//additionally define the meta-information
ito::HWMeta *meta = new ito::HWMeta("SerialIO"); //restriction to plugins with name "SerialIO"
param.setMeta(meta, true); //takes ownership of meta-pointer. Do not delete meta from the stack

//returns the hardware pointer casted to ito::AddInBase*
ito::AddInBase* value = param.getVal<ito::AddInBase*>();

ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric() //returns false
int len = param.getLen() //-1, since no length available
```

- **Reference to any initialized instance of DataObject (Type: DObjPtr)**

This is the reference to an instance of *DataObject*. The called method should check whether the instance has the necessary properties or type. Consider, that the flag *NoAutosave* is always set for that type. You can further give a meta information struct of class *DObjMeta* in order to specify the data object more in detail.

```
ito::DataObject *dObj = new ito::DataObject();
ito::Param param("image", ito::ParamBase::DObjPtr, dObj, "description");

//create a meta information where you only allow 2-dim data objects of type (u)int8.
//The necessary check is not automatically executed. You have to manually program it.
ito::DObjMeta *meta = new ito::DObjMeta(ito::tUInt8 | ito::tInt8, 2, 2);
param.setMeta(meta, true);

//returns the pointer casted to DataObject*
ito::DataObject* value = param.getVal<ito::DataObject*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric() //returns false
int len = param.getLen() //-1, since no length available

//if you do not need param again, you can delete dObj:
delete dObj;
dObj = NULL;
```

- **Reference to any initialized instance of ito::pclPointCloud (Type: PointCloudPtr)**

This is the reference to an instance of `pclPointCloud`. The called method should check whether the instance has the necessary properties or type. Consider, that the flag `typeNoAutosave` is always set for that type.

```
ito::Param param("pcl", ito::ParamBase::PointCloudPtr, NULL, "description");

//returns the pointer casted to pclPointCloud*
ito::pclPointCloud* value = param.getVal<ito::pclPointCloud*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()                //returns false
int len = param.getLen()                        //-1, since no length available
```

- **Reference to any initialized instance of `ito::pclPoint` (Type: `PointPtr`)**

This is the reference to an instance of `pclPoint`. The called method should check whether the instance has the necessary properties or type. Consider, that the flag `NoAutosave` is always set for that type. Currently, there is no meta information struct available for that type.

```
ito::Param param("point", ito::ParamBase::PointPtr, NULL, "description");

//returns the pointer casted to pclPoint*
ito::pclPoint* value = param.getVal<ito::pclPoint*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()                //returns false
int len = param.getLen()                        //-1, since no length available
```

- **Reference to any initialized instance of `ito::pclPolygonMesh` (Type: `PolygonMeshPtr`)**

This is the reference to an instance of `pclPolygonMesh`. The called method should check whether the instance has the necessary properties or type. Consider, that the flag `NoAutosave` is always set for that type. Currently, there is no meta information struct available for that type.

```
ito::Param param("polygonMesh", ito::ParamBase::PolygonMeshPtr, NULL, "description");

//returns the pointer casted to pclPolygonMesh*
ito::pclPolygonMesh* value = param.getVal<ito::pclPolygonMesh*>();
ito::RetVal retValue = param.setVal<void*>(ptr); //should return ito::retOk
bool numeric = param.isNumeric()                //returns false
int len = param.getLen()                        //-1, since no length available
```

ParamBase - Reference

`class ito::ParamBase`

Public Functions

ParamBase()

default constructor, creates “empty” `tParam`

ParamBase (const ByteArray & name)

constructor with name only creates a new *ParamBase* with name “name”, string is copied

Return

new *ParamBase* name “name”

Parameters

- name - name of new *ParamBase*

ParamBase (const ByteArray & name, const uint32 type)

constructor with name and type creates a new *Param* with name and type, string is copied

Return

new *Param* with name and type

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see *Type*

ParamBase (const ByteArray & *name*, const uint32 *type*, const char * *val*)

constructor with name and type, char val and optional info creates a new *ParamBase* with name, type, string value. Strings are copied

Return

new *ParamBase* with name, type, string value

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see *Type*
- `val` - character pointer to string pointer
- `info` - character pointer to string pointer holding information about this *ParamBase*

ParamBase (const ByteArray & *name*, const uint32 *type*, const double *val*)

constructor with name and type, double val creates a new *ParamBase* with name, type and val. Strings are copied.

Return

new *ParamBase* with name, type and val

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see *Type*
- `val` - actual value

ParamBase (const ByteArray & *name*, const uint32 *type*, const int *val*)

constructor with name and type and int val creates a new *ParamBase* with name, type and val

Return

new *ParamBase* with name, type and val.

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see *Type*
- `val` - actual value

ParamBase (const ByteArray & *name*, const uint32 *type*, const unsigned int *size*, const char * *values*)

array constructor with name and type, size and array creates a new *ParamBase* (array) with name, type, size and values.

Return

new *ParamBase* (array) with name, type, size and values.

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see *Type*
- `size` - array size

- `val` - values

ParamBase (const ByteArray & *name*, const uint32 *type*, const unsigned int *size*, const int * *values*)
array constructor with name and type, size and array creates a new *ParamBase* (array) with name, type, size and values.

Return

new *ParamBase* (array) with name, type, size and values.

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see Type
- `size` - array size
- `val` - values

ParamBase (const ByteArray & *name*, const uint32 *type*, const unsigned int *size*, const double * *values*)
array constructor with name and type, size and array creates a new *ParamBase* (array) with name, type, size and values.

Return

new *ParamBase* (array) with name, type, size and values.

Parameters

- `name` - name of new *ParamBase*
- `type` - type of new *ParamBase* for possible types see Type
- `size` - array size
- `val` - values

virtual **~ParamBase** ()

destructor

clear (frees) the name and in case a string value.

ParamBase (const *ParamBase* & *copyConstr*)
copy constructor creates *ParamBase* according to passed *Param*, strings are copied

Return

new *ParamBase* with copied values

Parameters

- `copyConstr` - *ParamBase* to copy from

const *ParamBase* **operator[]** (const int *num*) const

braces operator for element-wise access in arrays

braces operator returns the value of the index num from the array

Return

new tParam with values of *ParamBase*[num] in the array

Parameters

- `num` - array index for which the value should be returned

ParamBase & **operator=** (const *ParamBase* & *rhs*)

assignment operator (sets values of lhs to values of rhs tParam, strings are copied)

assignment operator sets values of lhs to values of rhs *ParamBase*, strings are copied

Return

new *ParamBase* with copied values

Parameters

- *rhs* - *ParamBase* to copy from

ito::RetVal **copyValueFrom** (const *ito::ParamBase* * *rhs*)

just copies the value from the right-hand-side tParam (*rhs*) to this tParam.

bool **isNumeric** (void) const

returns true if *Param* is of type char, int or double

bool **isValid** (void) const

returns whether tParam contains a valid type (true) or is an empty parameter (false, type == 0). The default tParam-constructor is always an invalid tParam.

uint32 **getType** (bool *filterFlags* = true) const

returns parameter type (autosave flag and optional flag are only included if filterFlags is set false)

uint32 **getFlags** (void) const

returns parameter flags (parameter type is not included)

uint32 **setFlags** (const uint32 *flags*)

sets parameter flags (parameter type remains untouched), for possible flags see tParamType

const char * **getName** (void) const

returns parameter name (string is not copied)

uint32 **getAutosave** (void) const

returns content of autosave flag - this flag determines whether the parameter value gets automatically saved to xml file when an instance of a plugin class is deleted (closed)

void **setAutosave** (const uint32 *autosave*)

sets content of autosave flag - this flag determines whether the parameter value gets automatically saved to xml file when an instance of a plugin class is deleted (closed)

template <typename _Tp>

ito::RetVal **setVal** (_Tp *val*)

setVal set parameter value - templated version

Return

RetVal with operation status sets the parameter value to the passed value, if the parameter type is inadequate it is set to the maximum value of template type

Parameters

- *val* - value to set to

template <typename _Tp>

ito::RetVal **setVal** (_Tp *val*, int *len*)

setVal set parameter value - templated version for arrays

Return

RetVal with operation status sets the parameter value to the passed value, if the length is below 1 or a Null pointer is passed an error is returned

Parameters

- *val* - value to set to
- *len* - length of array

template <typename _Tp>

_Tp getVal (void) const

getVal read parameter value - templated version returns the actual parameter value casted to the template parameter type. If the tParam has a non numeric type the largest value for the template type is passed.

Return

parameter value (numeric, casted)

template <typename _Tp>

_Tp getVal (int & *len*) const

getVal read parameter value - templated version for arrays returns the actual parameter value casted to the template parameter type. In 'len' is returned what is supposed to be the length of the array. As only array references are used within tParam the actual size may differ.

Parameters

- *len* - length of array

Param - Reference

class **ito::Param**

class for parameter handling e.g. to pass paramters to plugins

The plugins use this class to organize their parameters (internally) and for the paramList which is used for type checking whilst parsing parameters passed from python to c.

Public Functions

Param ()

default constructor, creates "empty" tParam

~Param ()

Destructor.

Param (const *Param* & *copyConstr*)

Copy-Constructor.

const *Param* **operator** [] (const int *num*) const

braces operator for element-wise access in arrays

Param & **operator=** (const *Param* & *rhs*)

assignment operator (sets values of lhs to values of rhs *Param*, strings are copied)

ito::RetVal **copyValueFrom** (const *ParamBase* * *rhs*)

just copies the value from the right-hand-side *ParamBase* (rhs) to this tParam.

const char * **getInfo** (void) const

< returns content of info string (string is not copied)

sets content of info string, if necessary the info buffer is freed first, passed string is copied

const *ParamMeta* * **getMeta** (void) const

returns const-pointer to meta-information instance or NULL if not available

ParamMeta * **getMeta** (void)

returns pointer to meta-information instance or NULL if not available

void **setMeta** (*ParamMeta* * *meta*, bool *takeOwnership* = false)

sets a new ParamMeta-instance as meta information for this *Param*

See

ito::ParamMeta

Parameters

- `meta` - is the pointer to any instance derived from *ParamMeta*
- `takeOwnership` - (default: false) defines, whether this *Param* should take the ownership of the *ParamMeta*-instance

double **getMin** () const

returns minimum value of parameter if this is available and exists.

This method is a wrapper method for `((ito::IntMeta*)getMeta())->getMax()`... and returns the minimum value of the underlying meta information. It only returns a valid value for meta structures of type `char`, `charArray`, `int`, `intArray`, `interval`, `range`, `double`, `doubleMeta`.

Return

minimum value of -Inf maximum does not exist

double **getMax** () const

returns maximum value of parameter if this is available and exists.

This method is a wrapper method for `((ito::IntMeta*)getMeta())->getMax()`... and returns the maximum value of the underlying meta information. It only returns a valid value for meta structures of type `char`, `charArray`, `int`, `intArray`, `range`, `double`, `doubleMeta`.

Return

maximum value of Inf maximum does not exist

8.3.5 Parameters - Meta Information

Every parameter of type `ito::Param` can contain meta information that describe some boundary values, value ranges, allowed values... of the parameter. Once a parameter has its valid meta information, itom is able to check given input values with respect to the meta information as well as adapt any auto-created input masks to simplify the input with respect to the given constraints.

Most possible types of class `ito::Param` have their respective meta information structure.

The base class of all kind of meta information classes is the class *ito::ParamMeta*. Any instance of class *ito::ParamMeta* can contain a pointer to an instance of this base class. If you know the type of parameter (e.g. `char`, `string` or `int`), you can safely cast this *ito::ParamMeta* base instance to the right meta information class that fits to the type.

Class ParamMeta

The class **ParamMeta** is the base class for all meta information classes. Parameters of class **Param** may contain pointers of that class, which then must be cast to the final implementation.

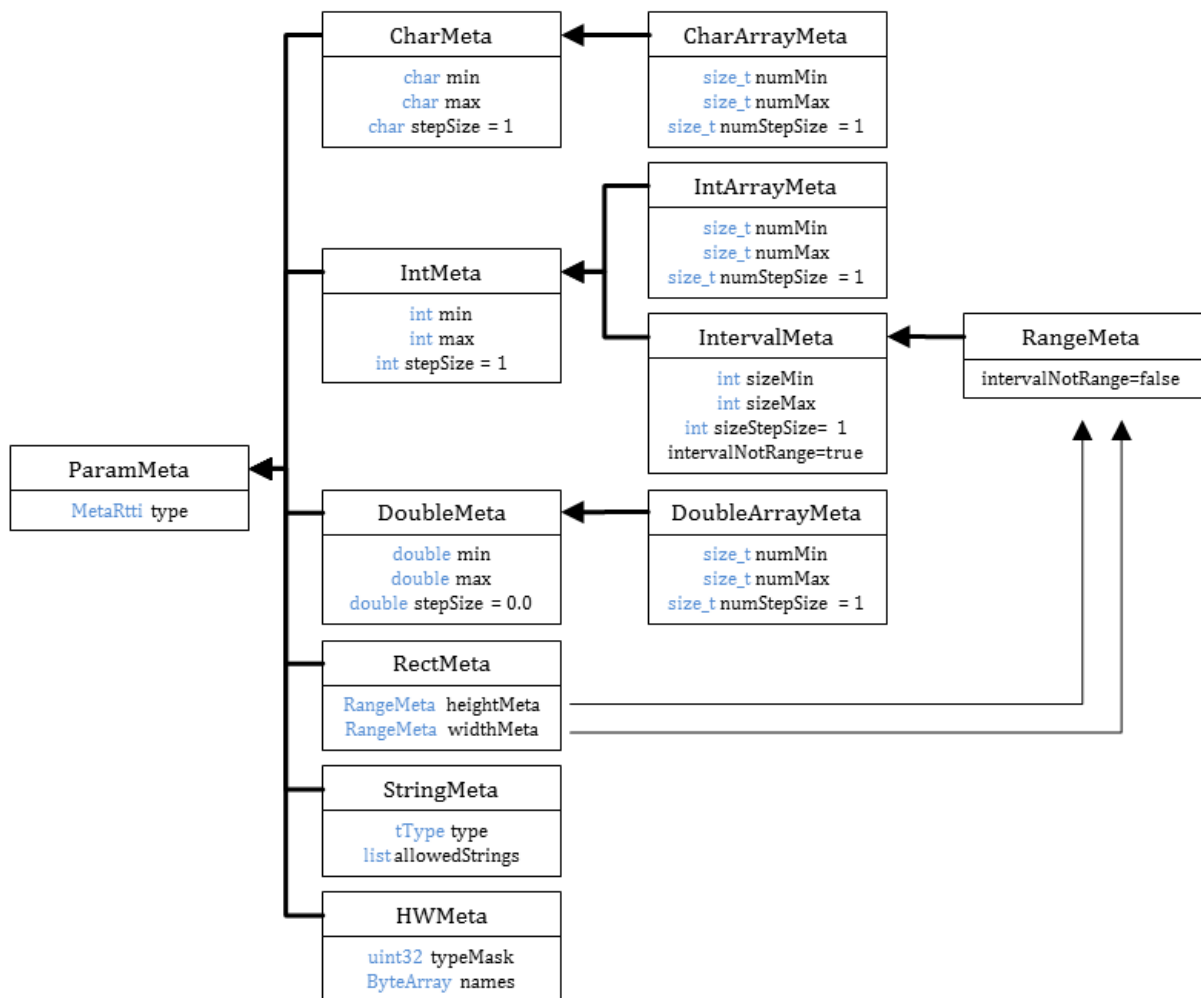
class **ito::ParamMeta**

Base class for all meta-information classes.

Parameters of type *ito::Param* can have a pointer to this class. Consider this base class to be abstract, such that it is only allowed to pass the right implementation (derived from this class) that fits to the type of the parameter. The runtime type information value `m_type` indicates the real type of this pointer, such that a direct cast can be executed.

See

ito::CharMeta, *ito::IntMeta*, *ito::DoubleMeta*, *ito::StringMeta*, *ito::HWMeta*, *ito::DObjMeta*,
ito::CharArrayMeta, *ito::IntArrayMeta*, *ito::DoubleArrayMeta*



Public Type

MetaRtti enum

Runtime type information.

MetaRtti is used to cast param meta objects, without having to enable runtime type information of the compiler.

Values:

- `rttiUnknown` = 0 - unknown parameter
- `rttiCharMeta` = 1 - meta for a char parameter
- `rttiIntMeta` = 2 - meta for an integer parameter
- `rttiDoubleMeta` = 3 - meta for a double parameter
- `rttiStringMeta` = 4 - meta for a string parameter
- `rttiHWMeta` = 5 - meta for a hardware plugin parameter
- `rttiDObjMeta` = 6 - meta for a data object parameter
- `rttiIntArrayMeta` = 7 - meta for an integer array parameter
- `rttiDoubleArrayMeta` = 8 - meta for a double array parameter
- `rttiCharArrayMeta` = 9 - meta for a char array parameter
- `rttiIntervalMeta` = 10 - meta for an integer array with two values that represent an interval [value1, value2] parameter
- `rttiDoubleIntervalMeta` = 11 - meta for a double array with two values that represent an interval [value1, value2] parameter (size of the interval is value2-value1)
- `rttiRangeMeta` = 12 - meta for an integer array with two values that represent a range [value1, value2] parameter (size of a range is 1+value2-value1)
- `rttiRectMeta` = 13 - meta for an integer array with four values that consists of two ranges (vertical and horizontal, e.g. for ROIs of cameras)

Public Functions

ParamMeta ()

default constructor with an unknown meta information type

ParamMeta (MetaRtti type)

constructor used by derived classes to indicate their real type

virtual ~ParamMeta ()

destructor

MetaRtti getType () const

returns runtime type information value

Class CharMeta, IntMeta and DoubleMeta

The classes **CharMeta**, **IntMeta** and **DoubleMeta** provide meta information for parameters of single numeric types.

class ito::CharMeta

meta-information for *Param* of type Char.

An object of this class can be used to parametrize a parameter whose type is `ito::ParamBase::Char`. If set, the given char number can be limited with respect to given minimum and maximum values as well as an optional step size (default: 1).

See

ito::Param, ito::ParamMeta

Public Functions

CharMeta (char *minVal*, char *maxVal*, char *stepSize* = 1)

constructor with minimum and maximum value

constructor with minimum and maximum value as well as optional step size (default: 1)

char **getMin** () const

returns minimum value

char **getMax** () const

returns maximum value

char **getStepSize** () const

returns step size

void **setMin** (char *val*)

sets the minimum value

Parameters

- *val* - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to *val*, too

void **setMax** (char *val*)

sets the maximum value

Parameters

- *val* - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to *val*, too

void **setStepSize** (char *val*)

sets the step size

Parameters

- *val* - is the new step size, hence only discrete values [*minVal*, *minVal*+*stepSize*, *minVal*+2**stepSize*...,*maxVal*] are allowed

Public Static Functions

CharMeta * **all** ()

returns a new instance of *CharMeta*, where the min and max are set to the full range available for char.

class **ito::IntMeta**

Meta-information for *Param* of type Int.

An object of this class can be used to parametrize a parameter whose type is *ito::ParamBase::Int*. If set, the given integer number can be limited with respect to given minimum and maximum values as well as an optional step size (default: 1).

See

ito::Param, ito::ParamMeta

Public Functions

IntMeta (int *minVal*, int *maxVal*, int *stepSize* = 1)
 constructor with minimum and maximum value as well as optional step size (default: 1)

int **getMin** () const
 returns minimum value

int **getMax** () const
 returns maximum value

int **getStepSize** () const
 returns step size

void **setMin** (int *val*)
 sets the minimum value

Parameters

- *val* - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to *val*, too

void **setMax** (int *val*)
 sets the maximum value

Parameters

- *val* - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to *val*, too

void **setStepSize** (int *val*)
 sets the step size

Parameters

- *val* - is the new step size, hence only discrete values [*minVal*, *minVal*+*stepSize*, *minVal*+2**stepSize*...,*maxVal*] are allowed

Public Static Functions

IntMeta * **all** ()
 returns a new instance of *IntMeta*, where the min and max are set to the full range available for integers.

class **ito::DoubleMeta**

Meta-information for *ito::Param* of type Double.

An object of this class can be used to parametrize a parameter whose type is *ito::ParamBase::Double*. If set, the given double number can be limited with respect to given minimum and maximum values as well as an optional step size (default: 0.0 -> no step size).

See

ito::Param, *ito::ParamMeta*

Public Functions

DoubleMeta (double *minVal*, double *maxVal*, double *stepSize* = 0.0)
 constructor with minimum and maximum value

double **getMin** () const
 returns minimum value

double **getMax** () const
returns maximum value

double **getStepSize** () const
returns step size

void **setMin** (double *val*)
sets the minimum value

Parameters

- *val* - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to *val*, too

void **setMax** (double *val*)
sets the maximum value

Parameters

- *val* - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to *val*, too

void **setStepSize** (double *val*)
sets the step size

Parameters

- *val* - is the new step size, hence only discrete values [*minVal*, *minVal*+*stepSize*, *minVal*+2**stepSize*...,*maxVal*] are allowed

Public Static Functions

DoubleMeta * **all** ()

returns a new instance of *DoubleMeta*, where the min and max are set to the full range available for double.

Class CharArrayMeta, IntArrayMeta and DoubleArrayMeta

The classes **CharArrayMeta**, **IntArrayMeta** and **DoubleArrayMeta** provide meta information for array-based parameters of numeric types. These classes are derived from **CharArray**, **IntArray** or **DoubleArray**, such that the minimum and maximum value as well as the step size for each single value is given by the features of their base class. Additionally, it is possible to set a min, max and *stepSize* constraint concerning the number of elements of the arrays.

class **ito::CharArrayMeta**

Meta-information for *Param* of type *CharArrayMeta*.

Meta-information for *Param* of type *IntArrayMeta*.

An object of this class can be used to parametrize a parameter whose type is *ito::ParamBase::CharArray*. Since this meta information class is derived from *ito::CharMeta*, it is possible to restrict each value to the single value constraints of *ito::CharMeta*. Furthermore, this class allows restricting the minimum and maximum length of the array as well as the optional step size of the array's length.

An object of this class can be used to parametrize a parameter whose type is *ito::ParamBase::IntArray*. Since this meta information class is derived from *ito::IntMeta*, it is possible to restrict each value to the single value constraints of *ito::IntMeta*. Furthermore, this class allows restricting the minimum and maximum length of the array as well as the optional step size of the array's length.

See

ito::Param, *ito::ParamMeta*, *ito::CharMeta*

See

ito::Param, *ito::ParamMeta*, *ito::IntArray*

Public Functions

size_t **getNumMin** () const
returns minimum number of values

size_t **getNumMax** () const
returns maximum number of values

size_t **getNumStepSize** () const
returns step size of number of values

void **setNumMin** (size_t *val*)
sets the minimum number of values

Parameters

- *val* - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to *val*, too

void **setNumMax** (size_t *val*)
sets the maximum number of values

Parameters

- *val* - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to *val*, too

void **setNumStepSize** (size_t *val*)
sets the step size of the number of values

Parameters

- *val* - is the new step size, hence only discrete values [*minVal*, *minVal*+*stepSize*, *minVal*+2**stepSize*...,*maxVal*] are allowed

class **ito::IntArrayMeta**

Public Functions

size_t **getNumMin** () const
returns minimum number of values

size_t **getNumMax** () const
returns maximum number of values

size_t **getNumStepSize** () const
returns step size of number of values

void **setNumMin** (size_t *val*)
sets the minimum number of values

Parameters

- *val* - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to *val*, too

void **setNumMax** (size_t *val*)
sets the maximum number of values

Parameters

- `val` - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to `val`, too

void **setNumStepSize** (size_t *val*)
sets the step size of the number of values

Parameters

- `val` - is the new step size, hence only discrete values [`minVal`, `minVal+stepSize`, `minVal+2*stepSize`...,`maxVal`] are allowed

class **ito::DoubleArrayMeta**

Meta-information for *Param* of type *DoubleArrayMeta*.

An object of this class can be used to parametrize a parameter whose type is `ito::ParamBase::DoubleArray`. Since this meta information class is derived from `ito::DoubleArray`, it is possible to restrict each value to the single value constraints of `ito::DoubleArray`. Furthermore, this class allows restricting the minimum and maximum length of the array as well as the optional step size of the array's length.

See

ito::Param, *ito::ParamMeta*, *ito::DoubleMeta*

Public Functions

size_t **getNumMin** () const
returns minimum number of values

size_t **getNumMax** () const
returns maximum number of values

size_t **getNumStepSize** () const
returns step size of number of values

void **setNumMin** (size_t *val*)
sets the minimum number of values

Parameters

- `val` - is the new minimum value, if this is bigger than the current maximum value, the maximum value is changed to `val`, too

void **setNumMax** (size_t *val*)
sets the maximum number of values

Parameters

- `val` - is the new maximum value, if this is smaller than the current minimum value, the minimum value is changed to `val`, too

void **setNumStepSize** (size_t *val*)
sets the step size of the number of values

Parameters

- `val` - is the new step size, hence only discrete values [`minVal`, `minVal+stepSize`, `minVal+2*stepSize`...,`maxVal`] are allowed

Class StringMeta

By this meta information you can give information about restrictions of strings to different strings. These strings can be interpreted as pure strings, as wildcard-expressions or regular expressions. The corresponding checks must be defined manually. If a string-parameter has an enumeration defined, where the strings are interpreted as strings,

and if this parameter will automatically be parsed by any input mask in the GUI, the corresponding input text box becomes a drop-down menu with the given enumeration elements.

class **ito::StringMeta**

Meta-information for *Param* of type String.

An object of this class can be used to parametrize a parameter whose type is `ito::ParamBase::String`. If set, it is possible to restrict the a given string to fit to a given list of strings. This list of strings might be interpreted in an exact way (`tType::String`), as wildcard expressions (`tType::Wildcard`) or as regular expressions (`tType::RegExp`).

See

ito::Param, *ito::ParamMeta*

Public Functions

StringMeta (*tType type*)

constructor

Returns a meta information class for string-types.

See

tType

Parameters

- *type* - indicates how the string elements should be considered

StringMeta (*tType type*, *const char * val*)

constructor

Returns a meta information class for string-types.

See

tType

Parameters

- *type* - indicates how the string elements should be considered
- *val* - adds a first string to the element list

StringMeta (*const StringMeta & cpy*)

copy constructor

virtual **~StringMeta** ()

destructor

tType **getStringType** () *const*

returns the type how strings in list should be considered.

See

tType

int **getLen** () *const*

returns the number of string elements in meta information class.

*const char ** **getString** (*int idx = 0*) *const*

returns string from list at index position or NULL, if index is out of range.

bool **addItem** (*const char * val*)

adds another element to the list of patterns.

StringMeta & **operator+=** (*const char * val*)

add another pattern string to the list of patterns.

Class DObjMeta

This meta information class provides further information about allowed types and boundaries concerning the dimension of a data object.

class **ito::DObjMeta**

Meta-information for *Param* of type DObjPtr.

(not used yet)

See

ito::Param, ito::ParamMeta

Public Functions

int **getMinDim** () const

returns maximum allowed dimensions of data object

int **getMaxDim** () const

returns minimum number of dimensions of data object

Class HWMeta

By that implementation of a meta information class you can provide information about references to other instantiated plugins. Every plugin is defined by a bitmask of enumeration **ito::tPluginType** (defined in **addInActor.h**). You can either add a minimum bitmask, that is required, to the **HWMeta**-instance or you can define an exact name of a plugin, which must be met.

class **ito::HWMeta**

Meta-information for *Param* of type HWPtr.

An object of this class can be used to parametrize a parameter whose type is *ito::ParamBase::HWPtr*, that is an instance of another hardware plugin. If set, it is possible to restrict the given hardware plugin to a specific type (e.g. dataIO, dataIO + grabber, actuator...) and/or to limit it to a specific name of the plugin (e.g. SerialIO).

See

ito::Param, ito::ParamMeta

Public Functions

HWMeta (uint32 *minType*)

constructor

creates HWMeta-information struct where you can pass a bitmask which consists of values of the enumeration *ito::tPluginType*. The plugin reference of the corresponding *Param* should then only accept plugins, where all bits are set, too.

See

ito::Plugin, ito::tPluginType

HWMeta (const char * *HWAddInName*)

constructor

creates HWMeta-information struct where you can pass a specific name of a plugin, which only is allowed by the corresponding plugin-instance.

See

ito::Plugin

uint32 **getMinType** () const
returns type-bitmask which is minimally required by plugin-reference. Default 0.

See

ito::tPluginType

ito::ByteArray **getHWAddInName** () const
returns name of specific hardware plugin

8.3.6 Parameters - Validation

If a default-parameter (or let's say template) is given in form of an instance of class `ito::Param` and a new value in terms of an instance of class `ito::ParamBase` is given, you might be interested if the new value fits to the type and the optional restrictions given by meta information of the template.

Therefore, the class **ParamHelper** (folder *helper*) provide method to validate `ParamBase`-instances with respect to a given meta information struct or to compare the compatibility of two different parameters.

Note: Since **ParamHelper** is not directly available for plugins, the most important methods are also made available by the **API** functions (see *itom API*). Therefore the API-call for the validator functions is also indicated below.

Validation with respect to given meta information

If you have access to any meta information instance (derived from class **ParamMeta**), you can check whether the value of an instance of class **ParamBase** or its derived class **Param** fits to the given requirements. There are different validator functions depending on the type of meta information.

static ito::RetVal **validateCharMeta** (const ito::CharMeta *meta, double value)

This methods checks whether the number 'value' does not exceed the boundaries given by the char meta information 'meta'. If this is not the case *retError* with an appropriate error message is returned, else *retOk*.

API-Call: ito::RetVal **apiValidateCharMeta** (const ito::CharMeta *meta, double value)

static ito::RetVal **validateIntMeta** (const ito::IntMeta *meta, int value)

This methods checks whether the number 'value' does not exceed the boundaries given by the integer meta information 'meta'. If this is not the case *retError* with an appropriate error message is returned, else *retOk*.

API-Call: ito::RetVal **apiValidateIntMeta** (const ito::IntMeta *meta, int value)

static ito::RetVal **validateDoubleMeta** (const ito::DoubleMeta *meta, double value)

This methods checks whether the number 'value' does not exceed the boundaries given by the double meta information 'meta'. If this is not the case *retError* with an appropriate error message is returned, else *retOk*.

API-Call: ito::RetVal **apiValidateDoubleMeta** (const ito::DoubleMeta *meta, double value)

static ito::RetVal **validateDoubleMetaAndRoundToStepSize** (const ito::DoubleMeta *meta,
double &value, bool allowRound-
ing = true)

This methods checks whether the number 'value' does not exceed the boundaries given by the double meta information 'meta'. If this is not the case *retError* with an appropriate error message is returned, else *retOk*. In difference to *validateDoubleMeta()*, this method rounds the given value to the nearest value, that fits to the step size constraints (if a step size != 0.0 is given).

The rounding is only done, if *allowRounding* is equal to *true*, else *value* will not be modified and can be considered to be constant.

API-Call: Right now, there is no direct API-function for this function, however *validateAndCastParam()* uses this function for double-based parameters.

```
static ito::RetVal validateDoubleMetaAndRoundToStepSize (const ito::DoubleMeta *meta,
                                                         ito::ParamBase &doubleParam,
                                                         bool allowRounding = true)
```

This method is equal to the method described above. The single difference is that the value to test is not passed as single double value but as parameter that must have the type `ito::ParamBase::Double`.

```
static ito::RetVal validateStringMeta (const ito::StringMeta *meta, const char* value, bool mandatory = false)
```

This methods checks whether the string, passed by argument *value*, fits to the requirements of the string meta information *meta*. If it does not fit, *retError* is returned with an appropriate error message. If argument *mandatory* is false, *retOk* is also returned if the string is not given, hence, *value* is an empty string.

API-Call: `ito::RetVal apiValidateStringMeta (const ito::StringMeta meta, const char value, bool mandatory = false)`

```
static ito::RetVal validateHWMeta (const ito::HWMeta *meta, ito::AddInBase *value, bool mandatory = false)
```

This method checks whether the plugin given by ‘value’ fits to the requirements possibly defined in the ‘meta’-plugin meta information struct. If this is the case *retOk* is returned, else *retError* with an appropriate error message. If ‘value’ is NULL *retOk* is only returned if argument ‘mandatory’ is *false*.

API-Call: `ito::RetVal apiValidateHWMeta (const ito::HWMeta *meta, ito::AddInBase *value, bool mandatory = false)`

```
static ito::RetVal validateCharArrayMeta (const ito::ParamMeta *meta, const char* values, size_t len)
```

This methods checks whether the array of character values named ‘values’ (length ‘len’) fits to the requirements given by the meta information of *meta*. This can be both restrictions with respect to every single values as well as the length of the array. If this is not the case *retError* with an appropriate error message is returned, else *retOk*.

API-Call: `ito::RetVal apiValidateCharArrayMeta (const ito::ParamMeta meta, const char values, size_t len)`

```
static ito::RetVal validateIntArrayMeta (const ito::ParamMeta *meta, const int* values, size_t len)
```

This methods checks whether the array of integer values named ‘values’ (length ‘len’) fits to the requirements given by the meta information of *meta*. This can be both restrictions with respect to every single values as well as the length of the array. If this is not the case *retError* with an appropriate error message is returned, else *retOk*.

If the meta information indicates that an interval, range or rectangle is expected, the specific validations and tests are done, too.

API-Call: `ito::RetVal apiValidateIntArrayMeta (const ito::ParamMeta meta, const int values, size_t len)`

```
static ito::RetVal validateDoubleArrayMeta (const ito::ParamMeta *meta, const double* values, size_t len)
```

This methods checks whether the array of double values named ‘values’ (length ‘len’) fits to the requirements given by the meta information of *meta*. This can be both restrictions with respect to every single values as well as the length of the array. If this is not the case *retError* with an appropriate error message is returned, else *retOk*.

If the meta information indicates that an interval is expected, the specific validations and tests are done, too.

API-Call: `ito::RetVal apiValidateDoubleArrayMeta (const ito::ParamMeta meta, const double values, size_t len)`

Overall validation methods

The following functions are overall functions that use the methods described above, depending on the type of the given template and parameter.

```
static ito::RetVal validateParam (const ito::Param &templateParam, const ito::ParamBase &param,
                                   bool strict = true, bool mandatory = false)
```

This method uses the methods above to check whether the value of ‘param’ is valid with respect to the

type and meta information of 'templateParam'. If 'strict' is *false*, the type is tried to be converted to type of 'templateParam' if possible. The 'mandatory' parameter is redirected to the corresponding validation methods above as therefore has the same meaning.

API-Call: ito::RetVal ****apiValidateParam****(-same arguments-)

```
static ito::RetVal validateAndCastParam(const ito::Param &templateParam, ito::ParamBase
                                     &param, bool strict = true, bool mandatory = false, bool
                                     roundToSteps = false)
```

This method uses the methods above to check whether the value of 'param' is valid with respect to the type and meta information of 'templateParam'. If 'strict' is *false*, the type is tried to be converted to type of 'templateParam' if possible. The 'mandatory' parameter is redirected to the corresponding validation methods above as therefore has the same meaning.

The difference to `validateParam()` is that the given argument *param* can be changed to the same type than *templateParam* if they can be cast and if it is a double value, its real value can be adapted to fit any step size constraints (only if *roundToSteps* = *true*).

API-Call: ito::RetVal **apiValidateAndCastParam** (-same arguments-)

The real documentation about the structure of plugins is organized as follows:

8.3.7 Introduction to plugins

The software **itom** obtains most functionality by mainly two concepts. On the one hand there is the python scripting language, which allows you to use almost all available python modules that are available for python 3.2 or higher. On the other hand, the entire measurement system becomes powerful by the possibility to enhance its functionality by several plugins.

These plugins are separated into three main groups:

- Type **actuator**: Plugins of this basic type should be used if you want to connect any actuator, like motor stages, piezo actuators, focussing systems, ... to **itom** (see [Plugin class - Basic information](#) and [Plugin class - Actuator](#))
- Type **dataIO**: Plugins of this basic type should be used for connecting any input or output device to **itom**. The main representative of this group are cameras as input device or the serial port as an input/output device (see [Plugin class - Basic information](#) and [Plugin class - DataIO](#)). This group is subdivided into the following sub-types:
 - **grabber** for cameras (Please consider that the class of the camera-plugin should not directly derive from *AddInDataIO* but from *AddInGrabber*, which is derived from the first.
 - **ADDA** for any analog-digital converters
 - **rawIO** for further input-output devices, like display windows for SLM or LCoS-sensors, serial ports or plugins which do not fit to any other group, since the type **dataIO** is the most flexible plugin type.
- Type **algo**: Plugins of this type mainly contain different algorithms and/or advanced user interfaces like dialogs, main windows, widgets, ... (see [Plugin class - Basic information](#) and [Plugin class - Algo](#))

Each plugin is a different project in your programming environment and is finally compiled as shared library (DLL on Windows).

Plugin load mechanism of itom

The **itom**-base directory contains a folder **plugins**. This folder itself usually consists of different subfolders each having the name of a specific plugin. The folder can then contain a release and/or debug-version of the specific plugin DLL as well as further files which are necessary for running the plugin. If your plugin is dependent on other files, please consider to read the specific information about how to publish dependencies of each plugin.

At the startup of **itom**, the application recursively scans the **plugins** folder and looks for any *DLL*-file on Windows machines or *a*-file on a Linux operating system. Then each DLL is tried to be loaded using the plugin system provided by the **Qt**-framework. The *DLL* can successfully be loaded if the following prerequisites are fulfilled:

- The plugin is a release version if **itom** is started in release mode OR
- The plugin is a debug version (this can for example be seen if the DLL-name ends with *...d.dll*) if **itom** is started in debug mode
- The plugin is compiled using the same major and minor version of **Qt** than **itom** (it is possible to load a plugin compiled with **Qt** 4.8.3 with **itom** compiled with 4.8.2)
- The plugin is compiled with the same compiler than **itom**
- If the plugin is dependent on other shared libraries which are not linked using a delay-load mechanism, the plugin can only be loaded if every necessary shared library can be found and successfully be loaded. If the dependency could not be loaded, the plugin-load fails with an error message *module could not be loaded*.
- The remarks contained in the plugin with respect to a minimum and maximum version number of **itom** must correspond to the version number of your **itom**
- The plugin must be compiled with the same version string of the class **ito::AddInInterface** than the version contained in **itom** (this is not the general version of **itom**). The version string of **AddInInterface** can be seen at the end of the file **addInInterface.h** in the **common**-folder.

An overview about the load status of all detected library files can be seen by calling the dialog **loaded plugins**, accessible by **itom**'s menu **help >> loaded plugins...**

Finally, every successfully loaded plugin is included in the dock-widget **Plugin** of **itom**.

Basic plugin structure

Every plugin consists at least of two classes, which are both derived from two different base classes. All possible base classes are provided in the file:

`addInInterface.h`

which is contained in the folder **include/common** of **itom**'s SDK. This folder contains further header files which can be used in every plugin and contain interfaces and helper libraries with useful functions for successfully and easily program a plugin. For using these files you need to link your plugin against the libraries **itomCommonLib** and **itomCommonQtLib**. Additionally **itom** provides an application programming interface (API) such that plugins can access important methods of **itom**. For more information see [itom API](#).

The two classes of the plugin are as follows:

1. Interface- or factory-class (derived from class **AddInInterfaceBase**)

This class must be derived from the class **AddInInterfaceBase** and is the communication tunnel between **itom** and the plugin itself using the plugin-framework of **Qt**. The plugin framework creates one single instance of this class when the plugin DLL is loaded (that means at startup of **itom**). Therefore this class is considered to be a singleton instance and since it is always loaded by **itom** even if it is not really needed, this class is kept small and only provides basic information about the plugin itself.

For further information about the structure of this interface class see [Plugin Interface Class](#).

2. Individual plugin class (derived from class **AddInDataIO**, **AddInGrabber**, **AddInActuator** or **AddInAlgo**)

This class is the main class of the plugin and should contain the main functionality of the plugin. Depending on the plugin type, this class is derived from any of the classes **AddInDataIO**, **AddInGrabber**, **AddInActuator** or **AddInAlgo**, which are also contained in the files mentioned above. All these classes internally are derived from the base class **AddInBase**, which is the most general class used for plugin handling and organization in **itom**. Please do not directly derive from **AddInBase**.

In the case of an actuator, a camera or any other IO-device, every opened device is represented by one individual instance of its corresponding plugin class. Hence, it is possible to have multiple instances of every class opened in **itom**. The creation and deletion of any instance is at first requested by the **AddInManager** class (an internal class of **itom**) which itself redirects this

request to the singleton instance of the interface class in the corresponding plugin (This is the interface class mentioned in point 1 above).

In the case of an algorithm-plugin, this class mainly contains a set of static methods, each being one individual algorithm or user interface. At startup of **itom** the singleton instance of the interface-class is created. Additionally, this individual plugin class also is instantiated once (singleton) at startup of itom and its internal *init*-method provides an overview (list) of all available algorithm and user-interface functions to **itom**. Additionally the default parameter sets for all algorithms and widget-methods are requested by **itom** and startup and are then cached in order to provide faster access in any subsequent function calls.

Further information about the common parts of the plugin class, independent on the plugin's type, see *Plugin class - Basic information*. For detailed information about the implementation of the different plugin types, see *Plugin class - DataIO*, *Plugin class - Actuator* or *Plugin class - Algo*.

Communication between itom, Python and each plugin

The communication to plugins of type **actuator** and **dataIO** is only possible by calling the public methods defined in the base classes **AddInActuator** or **AddInDataIO**. In Python, there exist two classes **dataIO** and **actuator**. Both have an interface that is analog to the corresponding interface **AddInActuator** or **AddInDataIO** in C++. Therefore, if a certain method of these classes is called in python, the call is redirected to the corresponding plugin-method. However, this call is executed across a thread-change, since both python and each plugin (besides the algorithm-plugins) “live” in their own thread.

8.3.8 Plugin interface class

Every **itom** plugin must consist of at least two classes. One class is the real plugin class, that represents one device. Therefore it is possible to create multiple instances of this class, hence, open multiple devices of the same plugin. The other class is a necessary structure in order to allow the communication between **itom** and the plugin. This is the so-called interface class, inherited from the abstract base class **AddInInterfaceBase**.

This base class is part of the **ito**-namespace and is defined in the file:

```
addInInterface.h
```

This file is located in the *common* directory of the include directory of the **itom** SDK. You need to link against the library **itomCommonQtLib** in the SDK.

In the main header file of your plugin (with the exemplary name **MyPlugin**), use the following demo code in order to create that class:

```
1 //myPlugin.h
2
3 #include "common/addInInterface.h" //adapt the path depending on the location of your plugin
4
5 class MyPluginInterface : public ito::AddInInterfaceBase
6 {
7     Q_OBJECT
8     Q_INTERFACES(ito::AddInInterfaceBase)
9     PLUGIN_ITOM_API
10
11     public:
12         MyPluginInterface(QObject *parent = 0);
13         ~MyPluginInterface() {};
14         ito::RetVal getAddInInst(ito::AddInBase **addInInst);
15
16     private:
17         ito::RetVal closeThisInst(ito::AddInBase **addInInst);
18 };
```


In the code example above, the macro directives `Q_OBJECT` and `Q_INTERFACES(ito::AddInInterfaceBase)` (lines 7+8) force the compiler in the pre-compilation step to create the necessary code (done by the **Qt** framework) such that the class fits to the **Qt**-plugin system and is able to communicate by the common signal-slot-system of **Qt**. Remember that every class which is finally derived from the `QObject`-class (like `AddInInterfaceBase` is, too) must have the `Q_OBJECT` macro defined.

The constructor `MyPluginInterface(...)`, defined in line 11 is called once by the AddIn-Manager of **itom** at startup in order to create the singleton instance of the class `MyPluginInterface`. In the body of this method you should provide basic information about your plugin (see section [The constructor of the plugin interface class](#)).

The destructor in line 12 usually does not require further implementation, such that the empty body already can be given in the header file.

Finally, there are also the methods `getAddInInst` and `closeThisInst` which are the most important methods. If an user or some other part of **itom** request an instance of this plugin (that means not an instance of the interface we are talking in this section, but of the real plugin), the AddInManager of **itom** calls the method `getAddInInst` of the corresponding interface class. Then this interface has to create an instance of the plugin and set the given double-pointer parameter to the pointer of this newly created instance.

Inversely, the AddInManager of **itom** will call `closeThisInst` of an interface in order to force the plugin interface class to delete the plugin instance, given by the `addInInst` parameter. This mechanism is usually used by so-called factory-classes. Therefore we can consider the interface class to be a factory for one or more instances of the plugin itself (For information about the plugin class see [Plugin class - Basic information](#)).

The constructor of the plugin interface class

In your main source file of your plugin you can implement the constructor of the plugin interface class in the following exemplary way:

```
MyPluginInterface::MyPluginInterface(QObject *parent)
{
    m_type          = ito::typeActuator; //or: ito::typeAlgo, ito::typeDataIO, ito::typeDataIO /
    setObjectName("MyPlugin"); //this is the name of the plugin how it appears in itom

    m_description   = QObject::tr("Description of MyPlugin");
    m_author        = "Author's name";
    m_version       = CREATEVERSION(0,1,0);
    m_minItomVer    = CREATEVERSION(1,0,0);
    m_maxItomVer    = MAXVERSION;

    m_autoLoadPolicy = ito::autoLoadKeywordDefined;
    m_autoSavePolicy = ito::autoSaveAlways;

    //initialize mandatory parameters for creating an instance of MyPlugin
    m_initParamsMand.append( ito::Param("param1", ito::Param::String, \
        "defaultValue", tr("translatable description").toLatin1().data() ) );
    ...

    //initialize optional parameters for creating an instance of MyPlugin
    m_initParamsOpt.append( ito::Param("optParam1", ito::Param::Int, 0, 10, 5, \
        tr("translatable description of optParam1").toLatin1().data() ) );
    ...
}
```

At first, the constructor consists of a section where you define basic information about the plugin itself. In the second part you will define a list of mandatory and optional parameters which are required if any user wants to create an instance of the plugin, e.g. the user wants to open a new camera or connect any motor.

Part 1 (Basic information):

`int m_type`

Type of this plugin. Possible types are an OR-combination of the enumeration `ito::tPluginType`:

- typeActuator for actuator-plugins
- typeDataIO | typeGrabber for cameras and other grabbing devices
- typeDataIO | typeADDA for any analog/digital converters
- typeDataIO | typeRawIO for any other input-output-devices, like serial ports, display windows...
- typeAlgo for a plugin providing algorithms, filters or any other methods as well as graphical user interfaces, dialogs, ... which enhance the functionality of **itom**

void **setObjectName** (const QString &name)

use this method to set the name of your plugin. This name should be simple and should not contain special characters, since it not only appears in the list of plugins but is also the string used for initializing a plugin by the python scripting language.

QString **m_description**

Give an advanced description of your plugin.

QString **m_author**

Use this string to denote the author(s) of this plugin

int **m_version**

This integer variable contains the version of your plugin. A version string always consists of a major, minor and patch value. All these values are combined in the integer variable and can be created using the macro **CREATEVERSION(major,minor,patch)** (defined in *sharedStructures.h*), where the values major, minor and patch are integer values, too.

int **m_minItomVer**

Use this variable to denote the minimum version number of **itom** which is necessary to run this plugin. If you don't have any specific minimum version, use the macro **MINVERSION**, defined in *sharedStructures.h* (folder *common*).

int **m_maxItomVer**

Use this variable to denote the maximum version number of **itom**. Versions higher than this value do not allow to run this plugin. If you don't care about any maximum version, use the macro **MAXVERSION**, defined in *sharedStructures.h* (folder *common*).

ito::tAutoLoadPolicy **m_autoLoadPolicy**

Depending on the value of this variable, the internal parameters of the plugin can be loaded from a *xml*-file and set after the plugin's *init*-method has been called. The possible values for that variable are given by the enumeration **ito::tAutoLoadPolicy** and are

```
enum tAutoLoadPolicy {
    autoLoadAlways          = 0x1, /*!< always loads xml file by addInManager */
    autoLoadNever           = 0x2, /*!< never automatically loads parameters from xml-file (default) */
    autoLoadKeywordDefined = 0x4 /*!< only loads parameters if keyword autoLoadParams=1 exists */
};
```

For more information about the loading and/or saving of plugin's parameters, see [Automatic loading and saving of plugin parameters](#).

ito::tAutoSavePolicy **m_autoSavePolicy**

Depending on the value of this variable, the internal parameters of the plugin can be saved to a *xml*-file at shutdown of a plugin instance. The possible values for that variable are given by the enumeration **ito::tAutoSavePolicy** and are

```
enum tAutoSavePolicy {
    autoSaveAlways          = 0x1, /*!< always saves parameters to xml-file at shutdown */
    autoSaveNever           = 0x2 /*!< never saves parameters to xml-file at shutdown (default) */
};
```

For more information about the loading and/or saving of plugin's parameters, see [Automatic loading and saving of plugin parameters](#).

bool `m_callInitInNewThread`

Usually, the plugin's init method, where for instance the hardware is started and initialized, is called in a new thread in order to keep the GUI reactive during the whole process. If you change this member from its default value **true** to **false**, **init** is executed in the main thread and afterwards the plugin is moved to the new thread. For more information, see [Plugin class - Basic information](#) or the [info box](#).

Part 2 (mandatory and optional parameters):

Note: This part is only important if you build plugins of the basic types **dataIO** or **actuator**, since only plugins of these types can have multiple parameters, hence, it is usefull to parametrize their constructors. For algorithm- or filter-plugins, you can let the vectors **m_initParamsMand** and **m_initParamsOpt** unchanged (hence empty).

If you create an instance of a plugin using the python language, you have mainly two possibilites:

- Plugins of type **dataIO** are addressed using the python type **dataIO**, which is a class of the module *itom*:

```
from itom import * # usually this import already has been done for you
variable = dataIO(PluginName,mandatoryParam1, ..., mandatoryParamN, optionalParam1, ...,
```

OR

```
import itom
variable = itom.dataIO(PluginName, mandatoryParam1, ..., optionalParam1, ...)
```

- Plugins of type **actuator*** are addressed using the python type **actuator**, which is a class of the module *itom*, too. The call is then analogous to the examples above:

```
variable = actuator(PluginName,mandatoryParam1, ..., mandatoryParamN, optionalParam1, ...)
```

- Plugins of type **algo** do not have any corresponding class in *itom*, since they are globally organized by **itom**. Algorithms can be called using the method `filter()`, windows, dialogs or further user interfaces provided by plugins are loaded using the static method `createNewPluginWidget()` of class *itom.uiDialog*.

The constructor of each plugin can have a list of mandatory and optional parameters, which must or can be provided if creating an instance of the plugin. Internally, each parameter is a value of type **Param**, which is a class of **itom** and provides values of different types. Each value has a specific name, a default value and a description string, which should be given or set to NULL. Additionally, depending on the parameters type, a minimum and maximum value can be indicated. For more information about class *Param* see [Parameter-Container class of itom](#).

The mandatory parameters are contained in the vector

```
QVector<ito::Param> m_initParamsMand
```

Using the methods *append* or *insert* you can add an arbitrary number of values (type *Param*) to this vector. The type *QVector* is a **Qt**-specific class which is similar to *std::vector*. The optional parameters are analogously contained in the vector

```
QVector<ito::Param> m_initParamsOpt
```

If one is creating an instance of the plugins, e.g. using the python commands above, **itom** is reading the given vector of mandatory of optional parameters. The first parameter of the constructors of the python class *itom.dataIO* or *itom.actuator* stands for the name of the plugin. The number of the following parameters must be equal or bigger than the length of the mandatory parameter vector. The first *n* parameters must exactly fit to the type, order and possible boundary values of the mandatory parameter vector. This vector is then copied and the values are replaced by the values given by the python-constructors.

If the following parameters in the constructor don't have any keywords, they must also fit to the types,... of the optional parameter vector. If there are not enough parameters given, the default value will be taken. Additionally, if the user gives keywords to the parameters, each parameter will be checked against its corresponding value in the optional parameter vector where keyword and parameter-name are equal. After the first parameter having a keyword no keyword-less parameters are accepted.

This is an example of creating a plugin with a set of parameters, where the last two parameters are tagged with their keywords:

```
variable = dataIO("MyPlugin",2.0,"test",delay=1000,file="C:\\test.dat")
```

After that the mandatory and optional parameter vectors are read, copied and that their values are replaced by the values given by the constructor, the instance of the plugin is created and the method **init** of the plugin class is called with the mandatory and optional parameter vector as argument. That's the basic way such a plugin instance is created and initialized.

The set of mandatory and optional parameters of each plugin, including their default, minimum and maximum value, their name and description string, can be returned in python using the method `itom.pluginHelp()`.

Method `getAddInInst` of the plugin interface class

As default implementation, you can copy the following code block for your implementation of the `getAddInInst`-method:

```
1 ito::RetVal MyPluginInterface::getAddInInst(ito::AddInBase **addInInst)
2 {
3     NEW_PLUGININSTANCE(MyPlugin)
4     return ito::retOk;
5 }
```

In case of an algorithm plugin use:

```
1 ito::RetVal MyPluginInterface::getAddInInst(ito::AddInBase **addInInst)
2 {
3     NEW_PLUGININSTANCE(MyPlugin)
4     REGISTER_FILTERS_AND_WIDGETS
5     return ito::retOk;
6 }
```

Since your plugin instance (**MyPlugin**) is finally derived from **AddInBase**, its private member **m_uniqueID** is automatically given an auto-incremented, unique number. Additionally if possible assign a string identifier that helps to identify the opened device. Set the identifier using the method **setIdentifier**. Please only set the identifier in the constructor or in the **init**-method of the plugin itself. The identifier is saved in the member **m_identifier**.

In the method above, it is assumed that your main class of your plugin *MyPlugin* is called *MyPlugin*, too. Then in line 3, a new instance of that class is created and this new instance is noticed about its own factory class in line 4. The factory class is hereby the pointer to this singleton interface class instance. Finally the given double pointer is set to the pointer of the newly created plugin instance. Finally, every plugin interface class has a protected member vector called *m_InstList* which contains a list of plugin instances opened by this interface (or factory). The newly created plugin is added to this list in line 6.

The return value of this method is of type **ito::RetVal**, which is set to the status **Ok**. For more information about the return value class **ito::RetVal** see [RetVal - The return type of itom methods](#).

Method `closeThisInst` of the plugin interface class

For this method, you can basically copy the following default implementation:

```
1 ito::RetVal MyPluginInterface::closeThisInst(ito::AddInBase **addInInst)
2 {
3     REMOVE_PLUGININSTANCE(MyPlugin)
4     return ito::retOk;
5 }
```

The **AddInManager** of **itom** is calling this method if the given plugin instance (parameter **addInInst**) should be deleted. If the parameter pointer is available, the plugin instance is removed from the list of loaded plugin instances (see [Method `getAddInInst` of the plugin interface class](#)) and the plugin instance is deleted.

8.3.9 AddInBase

8.3.10 Plugin class - Basic information

All plugins, especially hardware-related ones (derived from **AddInActuator**, **AddInDataIO**, ...), have some similar or identical base components. These are described in this chapter. Therefore it is recommended to read this chapter before continuing with the specific documentation about the type of plugin you want to create. The information given in this chapter is mainly important for plugins finally derived from classes **AddInActuator** (actuator-plugins) or **AddInDataIO** (displays, cameras, AD-converter, ...). Algorithm or filter-plugins behave a little bit different, however it remains worth to read this document first.

Common components for actuator and dataIO plugins

The following components and structures are equal for all plugins of type *actuator* or *dataIO* (including grabbers):

1. Every plugin has an internal set of parameters, that can be read or changed by the method **getParam** and **setParam** (class **AddInBase**).
2. These parameters are saved in the map **m_params**, member of class **AddInBase**.
3. Each parameter is an instance of class **ito::Param**.
4. The plugin itself is responsible if the new input to any parameter is valid or if the parameter is read-only as has to reject wrong input. For this purpose the parameters' meta information or flags can be used.
5. Every plugin executes its methods mainly in an own thread.
6. The communication between the plugin and its GUI-components (Config-Dialog, Dock-Widget in main window) must therefore be done using a signal-slot-connection or a simple invoke, that allows to call methods in another thread.
7. The plugin's **init** method is called with the mandatory and optional parameters, those default is set the in the plugin's interface and which are updated by the user input.

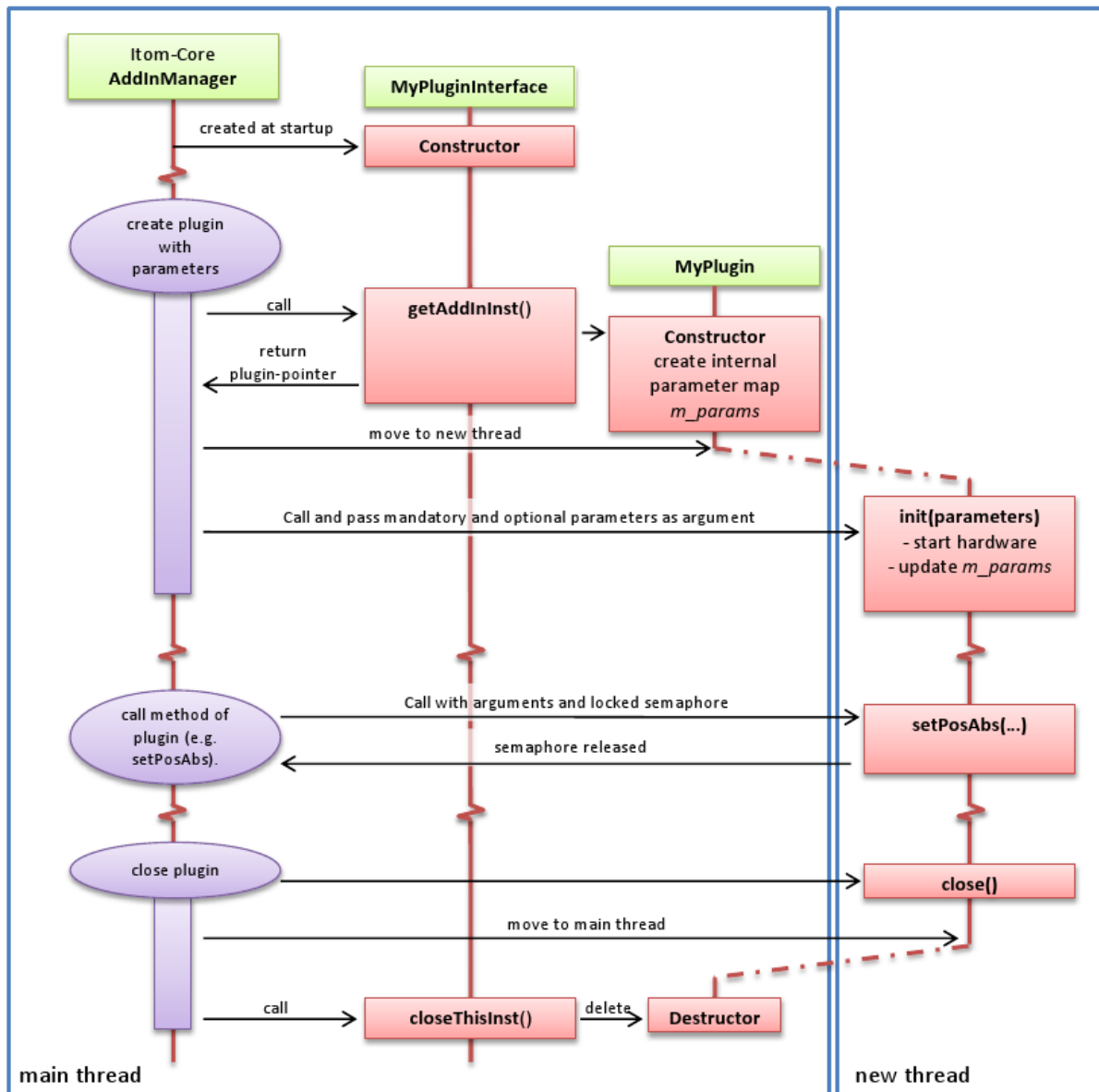
The basic scheme of the live-cycle of any plugin is depicted in the following figure:

At startup of **itom**, one single instance (singleton) of each plugin interface class is created and managed by the **AddInManager**. Later, an instance of the real plugin can be requested, for instance in a python-script or by a C-function call. The arguments to this request are two vectors of mandatory and optional parameters (instances of class **ParamBase**), that are passed to the initialization of the plugin. The default state of these vectors is obtained by the corresponding interface class (vectors **m_initParamsMand** and **m_initParamsOpt**). These default values have or can be replaced by the values, e.g. indicated in the python script (this is automatically done by **itom**).

Then **AddInManager** calls **getAddInInst()** of the plugin interface, that is creating a new instance of the plugin by calling its constructor. Please consider, that the mandatory and optional parameters are not passed to the constructor yet. Then, the pointer to the plugin is passed back to the **AddInManager**, that now moves the plugin to a newly created thread. From that point on, methods of the plugin should only be called using a thread-communication technique, provided by **Qt**, e.g. using the method **invokeMethod** or the signal-slot-system of **Qt**. Then the method **init** of the plugin is "invoked" by the **AddInManager** and the mandatory and optional initialization parameters are passed as arguments.

Next, the plugin can be used. Whenever a method of the plugin is called, either by python or any C-method, this call has to be executed again by an "over-thread" communication technique (**invokeMethod** or signal-slot). The last argument of such a call always is a locked instance of **ItomSharedSemaphore**. The caller then waits until the semaphore will be released in the plugin's method or a timeout occurred. Please make sure, that you release the semaphore in your plugin's method only at the point, when you don't need any parameters given by the caller any more. Usually it is released at the bottom of your method. Only in special cases (like method **waitForDone** for actuators) it makes sense to release it earlier. For more information about the semaphore see [ItomSharedSemaphore](#).

Finally, if the plugin should be closed, the inverse function calls with respect to the initialization are executed. That means, the **AddInManager** is at first invoking the method **call** of the plugin (still in its own thread). Then



the plugin is moved back to the main thread and then **AddInManager** forces the deletion of the plugin by the method **closeThisInst** of the corresponding interface class.

Note: In some cases, it is not possible to initialize the hardware (some cameras) in another thread than the main thread. Since this initialization should be done in the **init** method of the plugin, **itom** provides a possibility to call **init** before moving the plugin to the new thread. Hence, those both steps in *this scheme* are switched. This can be done by setting the member **m_callInitInNewThread** of the plugin interface class from its default value **true** to **false**. However, only use this possibility if there is no other chance, since the GUI is completely blocked during an initialization executed in the main thread (see also *Plugin interface class*).

Plugin-Framework

The bare framework of any plugin-class of type **DataIO**, **Actuator**, ... (but not **Algo**), looks like this:

```

1  #include "common/addInInterface.h"
2
3  class MyPlugin : public ito::AddInActuator OR public ito::AddInDataIO OR public ito::AddInGrabber
4  {
5      Q_OBJECT
6
7      public:
8          friend class MyPluginInterface;
9
10     protected:
11         ~MyPlugin() {};
12         MyPlugin();
13
14     public slots:
15         ito::RetVal init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt,
16         ito::RetVal close(ItomSharedSemaphore *waitCond);
17
18         ito::RetVal getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond = NULL);
19         ito::RetVal setParam(QSharedPointer<ito::ParamBase> val, ItomSharedSemaphore *waitCond = NULL);
20
21         ito::RetVal execFunc(const QString funcName, QSharedPointer<QVector<ito::ParamBase> > paramsMand,
22         ... QSharedPointer<QVector<ito::ParamBase> > paramsOpt, QSharedPointer<QVector<ito::ParamBase> > paramsMand,
23         ... ItomSharedSemaphore *waitCond = NULL);
24
25     private slots:
26         void dockWidgetVisibilityChanged(bool visible);
27 };

```

In the corresponding source file you need to write two defines before including the header file above:

```

1  #define ITOM_IMPORT_API
2  #define ITOM_IMPORT_PLOTAPI
3
4  #include "myPluginHeaderFromAbove.h"
5
6  //implement your code here

```

In this chapter, hints about implementing the methods in the definitions above are given. In the detailed chapters about every type of plugin, these class definitions will then be extended by the type-specific methods.

At first, your plugin class must be derived from class **ito::AddInActuator** if you want to create an actuator plugin, from class **ito::AddInGrabber** if you want to create a camera or grabber plugin and finally class **ito::AddInDataIO** for other hardware-related plugins. All those classes are derived from **AddInBase**. Next, the plugin's class must have the macro **Q_OBJECT** defined, such that the plugin is able to participate at the signal-slot-system of **Qt**. That is necessary for a multi-thread-communication with the plugin.

The constructor and destructor of the plugin are defined in the **protected** section. Only the interface class is able to call these methods, since the interface class is declared as friend of the plugin in line 6.

Then, there are five methods in the **public slots** section, which should be overwritten from their base definition in the class **AddInBase**. The methods **init** and **close** are important for the startup and shutdown of an instance of the plugin. **getParam** and **setParam** provide read and/or write access to parameters of the plugin, that are usually stored in the member-map **m_params** (however you can also use **getParam** and **setParam** to read and write other parameters. In some special cases, it is necessary to add further, arbitrary functions to the plugin, for instance in order to start an extra calibration process. These calls are executed by the general method **execFunc**, those first argument is the name of the function. You only have to implement this function, if you need this feature. Until now, this has only been necessary for a low amount of plugins.

Before the details about the implementation for these methods are explained, let us also take a look at the definition of the class **AddInBase** itself. Only few public or protected member functions of this class must be overwritten by the plugin (those, that have been described above). However, there are still a lot of further member methods which can be overwritten or useful for the functionality of your plugin. The description of the accessible member functions and attributes (public, protected, signals, public slots) is described in [AddInBase](#).

Now, let us give some hints about the implementation of the methods, that have to be overwritten:

Constructor

The constructor is a protected member method and should usually only be called by the corresponding interface class. A exemplary implementation is:

```

1 MyPlugin::MyPlugin(... further parameters ...) :
2     AddInActuator()
3 {
4     //create internal parameter map
5     ito::Param paramVal("name", ito::ParamBase::String | ito::ParamBase::ReadOnly, "MyPluginName");
6     m_params.insert(paramVal.getName(), paramVal);
7     //... add further parameters to map
8
9     //now create dock widget for this plugin (if available)
10    MyPluginDockWidget *myDockWidget = new MyPluginDockWidget(this);
11
12    //here: connect signals from the dock widget to the plugin
13    // connections that inform the dock widget about changes in parameters, status... should be c
14    // and destroyed in the method 'dockWidgetVisibilityChanged'.
15
16    //if you can and want, you can assign the unique identification string for this plugin here:
17    //m_idenfier = QString("my unique plugin nr: %1").arg(yourSerialNumber)
18    //you can also set this in the init-method.
19
20    Qt::DockWidgetAreas areas = Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea;
21    QDockWidget::DockWidgetFeatures features = QDockWidget::DockWidgetClosable | QDockWidget::Dock
22    createDockWidget("MyDockWidgetName", features, areas, myDockWidget);
23 }
```

For the constructor consider the following remarks:

1. The constructor is executed in the main thread and therefore is able to create widgets like the dock widget.
2. The constructor can have an arbitrary amount and type of parameters; the method **getAddInInst** of the interface class must be adapted to them.
3. Call the constructor of the base class (e.g. **AddInActuator**, **AddInDataIO**, **AddInGrabber**) in your constructor.
4. The constructor does not get the mandatory and optional parameters given by to user for initializing the plugin.
5. Create all internal parameters that are part of the parameter-map **m_params** and provide some default values.
6. If your plugin should provide a dock widget (that can be shown as dockable toolbox in the main window of **itom**), follow the snippet from the example above.

7. If your dock widget should be visible at initialization of the plugin or should be undocked as default state, overwrite and change the method **dockWidgetDefaultStyle** of class **AddInBase**.
8. See the specific documentation of each plugin to see which internal parameters must be available and which conventions exist for some specific parameters, that can be created.
9. If you want, set the unique identification string **m_identifier** in this constructor. You can also set it later in the **init**-method. If you don't set it, the auto-assigned unique ID is used for identifying your instance.

Destructor

Use the destructor (executed in main thread) for some final deletion work. Usually an empty body of the destructor is sufficient.

```
MyPlugin::~MyPlugin() {}
```

Init

The method **init** has the following bare framework:

```

1  ito::RetVal MyPlugin::init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue;
5
6      //use the content of paramsMand and paramsOpt (order and type
7      // with respect to m_initParamsMand and m_initParamsOpt of
8      // interface class) in order to initialize the hardware and
9      // change values of the m_params-map if necessary.
10
11     //if you want you can set the unique identification string here:
12     //m_identifier = QString("IAmAUniqueStringDescribingThisPlugin")
13
14     // emit signal about changed parameters
15     emit parametersChanged(m_params);
16
17     //release the wait condition and set its returnValue before
18     if (waitCond)
19     {
20         waitCond->returnValue = retValue;
21         waitCond->release();
22     }
23
24     setInitialized(true); //plugin is initialized
25     return retValue;
26 }
```

Consider the following hints for the implementation of the **init**-method:

1. Usually it is executed in a new thread (see [here](#))
2. The parameters are the filled mandatory and optional parameter vectors, those default implementation has been given in the constructor of the interface class.
3. You can be sure, that both the order and type of those plugins remains invariant.
4. Initialize the plugin with respect to these parameters.
5. If necessary change the value of the internal parameters **m_params**.
6. If so, emit the signal **parametersChanged(m_params)**.
7. At the end of the method release the wait condition.

8. Afterwards call **setInitialized(true)** in order to confirm the initialization of the plugin.
9. See the specific documentation of each plugin to see, what else has to be done in the method **init**.

Close

The method **close** is always executed as last method in the plugin thread. Disconnect your hardware in this method. A simple exemplary implementation is:

```

1  ito::RetVal MyPlugin::close(ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue;
5
6      //your code comes here
7
8      if (waitCond)
9      {
10         waitCond->release();
11         waitCond->returnValue = retValue;
12     }
13     return retValue;
14 }
```

getParam

This method is the getter-method for reading the current value of internal parameters, that usually are an item of the map **m_params**. In this method (like in others, too) methods provided by the **itom** API are used. Therefore, you need to include the API header in your source file, e.g. by

```
#include "common/apiFunctionsInc.h"
```

The prototype for the method *getParam* then looks like this:

```

1  ito::RetVal MyPlugin::getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue;
5      QString key;
6      bool hasIndex = false;
7      int index;
8      QString suffix;
9      QMap<QString, ito::Param>::iterator it;
10
11     //parse the given parameter-name (if you support indexed or suffix-based parameters)
12     retValue += apiParseParamName(val->getName(), key, hasIndex, index, suffix);
13
14     if(retValue == ito::retOk)
15     {
16         //gets the parameter key from m_params map (read-only is allowed, since we only want to g
17         retValue += apiGetParamFromMapByKey(m_params, key, it, false);
18     }
19
20     if(!retValue.containsError())
21     {
22         //put your switch-case.. for getting the right value here
23
24         //finally, save the desired value in the argument val (this is a shared pointer!)
25         //if the requested parameter name has an index, e.g. roi[0], then the sub-value of the
26         //array is split and returned using the api-function apiGetParam
27         if (hasIndex)
```

```

28     {
29         *val = apiGetParam(*it, hasIndex, index, retValue);
30     }
31     else
32     {
33         *val = *it;
34     }
35 }
36
37 if (waitCond)
38 {
39     waitCond->returnValue = retValue;
40     waitCond->release();
41 }
42
43 return retValue;
44 }

```

The **getParam** is either called directly, by changing the thread or by the **Python** script execution, if the appropriate method **getParam** in **Python** is called. For guaranteeing a thread-safe implementation, the first argument is a shared pointer to a value of class **ito::Param**. In case of success, the requested value must be saved into this given parameter. The second argument is a string containing the parameter name.

It is possible to pass a single parameter name or to follow a given string-syntax in order to also give a certain array index as well as an additional suffix string. Rules for this syntax are given in the section *Possible parameter names for parameter-map m_params*. In order to split the given raw name into its possible components **name**, **index** and **suffix**, the method **parseParamName** defined in **helperCommon.h** can be used. In the exemplary implementation above, the parameter-name is finally searched in the map **m_params** and if found, the value of this parameter is copied to the given argument **val**. However, this implementation is the most basic one and you can also implement further lines of code.

Since this method can also be called from another thread, an instance of **ItomSharedSemaphore** is passed, that finally needs to be released. The destruction of the pointer **waitCond** is done by the locker-instance **locker**.

setParam

The method **getParam** is the analog method to **setParam**. However, in **setParam** you will not only passed a parametername but also the new value for this parameter. Then you need to implement the following checks:

- Check whether your internal parameter is **read-only** and if so, reject the set-command.
- Verify the compatibility of the parameter-type of the new and internal parameter.
- Consider further restrictions given by a possible meta information that is appended to the internal parameter and check if the new value fits to these restrictions.
- If any internal parameter changed, emit the signal **parametersChanged** with **m_params** as argument. This signal is globally defined in the base class **AddInBase** and is for instance connected with GUI-elements like the possible dock-widget (toolbox) of this plugin.

Finally, an exemplary (simplified) version for the method **setParam** is:

```

1  ito::RetVal MyPlugin::setParam(QSharedPointer<ito::ParamBase> val, ItomSharedSemaphore *waitCond)
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4      ito::RetVal retValue(ito::retOk);
5      QString key;
6      bool hasIndex;
7      int index;
8      QString suffix;
9      QMap<QString, ito::Param>::iterator it;
10
11      //parse the given parameter-name (if you support indexed or suffix-based parameters)

```

```

12     retValue += apiParseParamName( val->getName(), key, hasIndex, index, suffix );
13
14     if(isMotorMoving()) //this if-case is for actuators only.
15     {
16         retValue += ito::RetVal(ito::retError, 0, tr("any axis is moving. Parameters cannot be set"));
17     }
18
19     if(!retValue.containsError())
20     {
21         //gets the parameter key from m_params map (read-only is not allowed and leads to ito::retError)
22         retValue += apiGetParamFromMapByKey(m_params, key, it, true);
23     }
24
25     if(!retValue.containsError())
26     {
27         //here the new parameter is checked whether its type corresponds or can be cast into the
28         // value in m_params and whether the new type fits to the requirements of any possible
29         // meta structure.
30         retValue += apiValidateParam(*it, *val, false, true);
31
32         //if you program for itom 1.4.0 or higher (Interface version >= 1.3.1) you should use this
33         //API method instead of the one above: The difference is, that incoming parameters that are
34         //compatible but do not have the same type than the corresponding m_params value are cast
35         //to the type of the internal parameter and incoming double values are rounded to the
36         //next value (depending on a possible step size, if different than 0.0)
37         retValue += apiValidateAndCastParam(*it, *val, false, true, true);
38     }
39
40     if(!retValue.containsError())
41     {
42         if(key == "async")
43         {
44             //check the new value and if ok, assign it to the internal parameter
45             retValue += it->copyValueFrom( &(*val) );
46         }
47         else if(key == "demoKey")
48         {
49             //check the new value and if ok, assign it to the internal parameter
50             retValue += it->copyValueFrom( &(*val) );
51         }
52         else
53         {
54             //all parameters that don't need further checks can simply be assigned
55             //to the value in m_params (the rest is already checked above)
56             retValue += it->copyValueFrom( &(*val) );
57         }
58     }
59
60     if(!retValue.containsError())
61     {
62         emit parametersChanged(m_params); //send changed parameters to any connected dialogs or d
63     }
64
65     if (waitCond)
66     {
67         waitCond->returnValue = retValue;
68         waitCond->release();
69     }
70
71     return retValue;
72 }

```

In this base implementation, the `parametername` is firstly searched in the `m_params`-map. If found, the internal parameter is checked for the **read-onlyness**. Next, the method distinguishes between numeric and non-numeric parameters, since the numeric-ones can possibly be casted from one numeric type to another one. If these prerequisites are met, you should then check the new value for certain restrictions and if this is met, too, the internal parameter can be filled with the content of the given, new value. Else, appropriate error messages should be returned and the semaphore **waitCond** must finally be released.

If you want to set multiple parameters in an uninterrupted sequence, you can also call or invoke the method **setParamVector** defined in **AddInBase**. You don't have to implement this method, since this is already done. For more infos about this method, see the definition of **AddInBase** ([AddInBase](#)).

Extend your plugin by registering further functions

As already mentioned above, you can extend the set of methods of each plugin, defined by their base classes, by registering further functions. This is for instance useful, if you want to provide methods for starting a gamma correction, starting a specific calibration, ...

Each such method has the same arguments than the **init**-method of the plugin:

- A vector of mandatory parameters (class **ParamBase**)
- A vector of optional parameters (class **ParamBase**)
- A Vector of out-only parameters (class **ParamBase**)

In the constructor of your plugin, you need to register each of those methods by also predefining the default set of all parameter-vectors (mandatory, optional, out). Only, if this step has been done, the corresponding method can be called, for instance by using the method **exec** in the **Python** scripting language. The call to this method finally leads to a call of the method **execFunc**, defined in the class **AddInBase**, which has to be overwritten in your plugin, if you want to provide at least one additional method.

The first parameter of **execFunc** is the name of the additional method. Therefore you need to check for this name and then execute the corresponding algorithm.

An exemplary implementation of the method **execFunc** is

```
1  ito::RetVal MyPlugin::execFunc(const QString funcName, QSharedPointer<QVector<ito::ParamBase> > p
2      ...QSharedPointer<QVector<ito::ParamBase> > paramsOpt, QSharedPointer<QVector<ito::ParamBase>
3  {
4      ito::RetVal retValue = ito::retOk;
5      ito::ParamBase *param1 = NULL;
6      ito::ParamBase *param2 = NULL;
7
8      if(funcName == "saveXMLParams")
9      {
10         param1 = ito::getParamByName(&(*paramsMand), "filename", &retValue);
11         param2 = ito::getParamByName(&(*paramsOpt), "overwriteIfExists", &retValue);
12
13         if(!retValue.containsError())
14         {
15             retValue += XmlParameter::saveXmlParams( param1->getVal<char*>(), m_xmlParams, static
16         }
17     }
18     else
19     {
20         retValue += ito::RetVal::format(ito::retError,0,tr("function name '%s' does not exist").t
21     }
22
23     if(waitCond)
24     {
25         waitCond->returnValue = retValue;
26         waitCond->release();
27         waitCond->deleteSemaphore();
28         waitCond = NULL;
```

```

29     }
30
31     return retValue;
32 }

```

In this method, one additional function (name **saveXMLParams**) is integrated, which only is called with one mandatory and optional parameter and no further output parameter. The corresponding registration of this method is integrated in the constructor of your plugin:

```

1  //register exec functions
2  QVector<ito::Param> pMand;
3  pMand << ito::Param("filename", ito::ParamBase::String | ito::ParamBase::In, NULL, tr("absolute f
4  QVector<ito::Param> pOpt;
5  pOpt << ito::Param("overwriteIfExists", ito::ParamBase::Int | ito::ParamBase::In, 0, 1, 1, tr("pa
6  QVector<ito::Param> pOut;
7  registerExecFunc("saveXMLParams", pMand, pOpt, pOut, tr("description"));

```

Possible parameter names for parameter-map **m_params**

The parameters which you add to the **m_params** map, must have a name which fits to the following rules:

- The name starts with a lower or upper case character, hence a value between *a-z* or *A-Z*
- The first character of the name can be followed by an infinite number of alpha-numerical characters (including characters like *_* or *-*).

If the user tries to get or set a parameter e.g. using the python-commands **setParam** or **getParam**, the parameter name firstly is checked if it exists in the **m_params** vector before the corresponding getter or setter methods in the plugin are called. However not the full given parameter string has to match any available key-name in the **m_params** map, since there are further information which can be encoded in the parameter name:

Sometimes it is useful to create a parameter as integer or double array, e.g. for the speed values of every single axis. In this case you will create a parameter with the name **speed** having a type **typeDoubleArray**. Then the user can either access the whole array using the real parameter name **speed**, or the user can access one single value of the array by appending the index within brackets at the parameter name. In this case, it is the programmers task to parse the given parameter string and separate the parameter's real name (here: **speed**) and the index. Then the index has to be checked with respect to its lower and upper bound.

As further possibility, it is also allowed to append further information to a parameter. This is done by appending a colon-character to the parameter name followed by any string, which is the additional information string.

To sum this description up, let us assume that the parameter in **m_params** has the name **PARAMNAME**, which fits to the rules above. Then **itom** will accept parameter names, which correspond to the following rules:

- **PARAMNAME**
- **PARAMNAME[INDEX]**
- **PARAMNAME[INDEX]:ADDITIONALTAG**
- **PARAMNAME:ADDITIONALTAG**

INDEX has to be any fixed-pointer number, the **ADDITIONALTAG** can be any string.

It is the programmer's responsibility to split the given parameter name in the three components **PARAMNAME**, **INDEX** and **ADDITIONALTAG** if the corresponding parameter has the ability to handle indexed-values or even additional information (encoded in **ADDITIONALTAG**).

In order to execute this split you can use the api-method

```
ito::RetVal apiParseParamName(const QString &name, QString &paramName, bool &hasIndex, int &index,
```

defined in **apiFunctionsInc.h**.

As an alternative you can use the following regular expression:

```
1 QString regularExpression = "^[a-zA-Z]+\\w* (\\[ (\\d+) \\]) {0,1} (: (.*) ) {0,1} $";
2 QRegExp rx(regularExpression)
3 if(rx.indexIn(yourString) == -1)
4 {
5     //yourString does not match the regular expression
6 }
7 else
8 {
9     QStringList components = rx.capturedTexts();
10    //components consists of the following sub-strings:
11    /*
12    [0] full string
13    [1] PARAMNAME
14    [2] [INDEX] or empty-string if no index is given
15    [3] INDEX or empty-string if no index is given
16    [4] :ADDITIONALTAG or empty-string if no tag is given
17    [5] ADDITIONALTAG or empty-string if no tag is given
18    */
19 }
```

8.3.11 Plugin class - DataIO

Base idea behind any DataIO

- Plugins of class **dataIO** operate systems that have input or output data, e.g. frame grabbers, cameras, serial ports, AD-DA-converters...
- Every single instance of class *dataIO* has an exclusive communication with one connected device. More cameras or ADDA-converter of the same type should be managed with a corresponding number of instances.
- Depending on the plugin, more than one DataIO can be separately controlled by the plug-number or the vendor/camera-ID.
- Like any other plugin, every DataIO has a set of parameters, which can be set/get using python by the commands **setParam** and **getParam**.
- Every DataIO is executed in its own thread.
- Every DataIO can have one configuration dialog and one toolbox (dockWidget).
- All parameters are stored in the member **m_params** of type QMap. They can be read or changed by methods **getParam** and **setParam**. Some parameters are read only!
- If parameters of DataIO change, they also must be updated in the **m_params**-map and the signal **parametersChanged** must be emitted.
- The data acquisition is performed according to the grabber subtype. These subtypes are 'typeGrabber', 'typeADDA' and 'typeRawIO'.

Grabber plugin

This is a subtype of DataIO for camera / framegrabber communication. Plugins of this type are inherited from **ito::AddInGrabber**. The data acquisition is managed as follows:

- The methods **startDevice** and **stopDevice** open and close the capture logic of the devices to reduce CPU-load. For serial ports these functions are unnecessary.
- The method **acquire** starts the DataIO grabbing a frame with the current parameters. The function returns after sending the trigger. The function should be callable several times without calling **getVal** or **copyVal**.
- The methods **getVal** and **copyVal** are the external interfaces for data grabbing. They call **retrieveData**. The function should not be callable without a previous call of **acquire** and than only once.

- In **retrieveData** the data transfer is done and frame has to be copied. The function blocks until the triggered data is copied. In case **retrieveData** is called by **getVal** the frame has to be copied to **m_data**, an internal **dataObject**.
- The function **getVal** overwrites the IO-**dataObject** by a shallow copy of the internal **dataObject**. Empty objects are allowed. Warning read shallow copy of **dataObject** before usage.
- The function **copyVal** deeply copies data to the externally given **dataObject**. The **dataObject** must have the right size and type. **dataObject** with ROI must not be overwritten. The ROI should be filled. Empty objects are allowed. In case of empty **dataObject** a new object with right size and type must be allocated.
- The internal **dataObject** is checked after parameter changes by **checkData** (size, size and bpp) and, if necessary, reallocated.

A typical sequence in python is

```

1 device.startDevice()
2 device.acquire()
3 device.getVal(dObj)
4 device.acquire()
5 device.getVal(dObj)
6 device.stopDevice()

```

A sample header file of the DataIO's plugin class might look like the following snippet:

```

1 #include "common/addInGrabber.h"
2 #include <qsharedpointer.h>
3
4 class MyCamera : public ito::AddInGrabber
5 {
6     Q_OBJECT
7
8     protected:
9         ~MyDataIO(); /*! < Destructor*/
10        MyDataIO(); /*! < Constructor*/
11
12        ito::RetVal retrieveData(ito::DataObject *externalDataObject = NULL); /*!< Wait for acqui
13
14    public:
15        friend class MyDataIOInterface;
16        const ito::RetVal showConfDialog(void); /*! Open the config nonmodal dialog to set cam
17        int hasConfDialog(void) { return 1; }; /*!< indicates that this plugin has got a configur
18
19    private:
20
21    public slots:
22        ito::RetVal getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond);
23        ito::RetVal setParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond);
24
25        ito::RetVal init(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItomSha
26        ito::RetVal close(ItomSharedSemaphore *waitCond);
27
28        ito::RetVal startDevice(ItomSharedSemaphore *waitCond);
29        ito::RetVal stopDevice(ItomSharedSemaphore *waitCond);
30
31        ito::RetVal acquire(const int trigger, ItomSharedSemaphore *waitCond = NULL);
32
33        ito::RetVal getVal(void *vpdObj, ItomSharedSemaphore *waitCond);
34
35        ito::RetVal copyVal(void *vpdObj, ItomSharedSemaphore *waitCond);
36
37    private slots:
38        void dockWidgetVisibilityChanged(bool visible);
39 };
40

```

```
41 class MyCameraInterface : public ito::AddInInterfaceBase
42 {
43     Q_OBJECT
44     #if QT_VERSION >= QT_VERSION_CHECK(5, 0, 0)
45     Q_PLUGIN_METADATA(IID "ito.AddInInterfaceBase" )
46     #endif
47     Q_INTERFACES(ito::AddInInterfaceBase)
48     PLUGIN_ITOM_API
49
50     protected:
51
52     public:
53         MyCameraInterface();
54         ~MyCameraInterface();
55         ito::RetVal getAddInInst(ito::AddInBase **addInInst);
56
57     private:
58         ito::RetVal closeThisInst(ito::AddInBase **addInInst);
59 };
```

Parameters and Unit Conventions

In order to have a unified behaviour of all camera plugins, respect the following unit conventions. That means, the plugin should store related parameters using these conventions, such that **getParam** and **setParam** returns and obtains values using these units. Internally, it is sometimes necessary to convert these units to the units required by the interface of the real camera device.

- Integration time, frame time... in **sec**
- bit depth / resolution in bit [8, 10, 12, 14, 16, 24]

Implement the following mandatory parameters in the map **m_params**:

- **“name”**: {string | readonly} name of the plugin
- **“bpp”**: {int} current bit depth (will be read e.g. when opening a live window)
- **“sizex”**: {int | readonly} current width of the camera image (considering a possible ROI). This parameter is always read-only and needs to be changed if the optional parameters *x0* or *x1* change. This parameter is read e.g. when a live window is opened.
- **“sizey”**: {int | readonly} current height of the camera image (considering a possible ROI). This parameter is always read-only and needs to be changed if the optional parameters *y0* or *y1* change. This parameter is read e.g. when a live window is opened.

If desired implement the following optional parameters in the map **m_params**:

- **“integration_time”**: {double} Exposure or integration time in seconds
- **“frame_time”**: {double} The time between two frames (in seconds, often read-only)
- **“gain”**: {double} Normalized gain in the range [0.0,1.0]
- **“offset”**: {double} Normalized offset in the range [0.0,1.0]
- **“x0”, “y0”**: {int, deprecated (see roi)} pixel coordinate of the left top corner of the image or ROI [0..width-1/height-1] If this changes, “sizex” or “sizey” must be changed, too.
- **“x1”, “y1”**: {int, deprecated (see roi)} pixel coordinate of the right bottom corner of the image or ROI [x0+1/y0+1..width-1/height-1] If this changes, “sizex” or “sizey” must be changed, too.
- **“roi”**: {int-array} Since itom AddIn Interface version 1.3.1 (itom 1.4.0 or higher), it is recommended to replace *x0* *y0*, *x1* and *y1* by the integer array based parameter **roi** which expects an array [left, top, width, height]. This parameter can easily be parametrized using the meta information `ito::RectMeta` and allows the direct configuration of the entire ROI or a single access to one of the four components, by passing the parametername `roi[0]`, `roi[1]`,....

AD-Converters

AD-Converter plugins are directly inherited from `ito::AddInDataIO`. An AD-DA-converter plugin has the following characteristics:

- It can communicate with 1 or multiple input and/or output channels. To set the number of total channels and to define if a channel is an incoming or outgoing channel, the plugin's parameters or initialization parameters should be used. Sometimes it is not possible to change the direction after initialization, this depends on the device.
- The method **startDevice** must be called like in a camera before the first usage of the device in order to establish the connection. Create a no-operation method, if this is not necessary for your device. It is possible, that startDevice is called multiple time, therefore count the number of starts and only establish the connection upon the first call.
- As counterpart to **startDevice**, the method **stopDevice** disconnects the device. For every call of **startDevice**, **stopDevice** must be called and at the last call, the connection should be interrupted.
- In difference to a camera dataIO plugin, the method **acquire** can be used to start the acquisition of a serie of input data values at all previously selected input channels. It is also possible to create an empty function here. Then the reading-process of new single data values for each input channel is totally executed in the method **getVal**.
- The method **copyVal** needs not to be implemented for AD-DA-converters.
- Method **getVal**: This method registers input values from all previously selected input channels (depending on your implementation and parametrization it is also possible to register multiple values per channel) and returns these values to the user.
 - If you have one or multiple input channels, use the definition **getVal(void *dObj, ItomSharedSemaphore *waitCond)**. The parameter dObj is then a pointer to `ito::DataObject` and can be cast to this class. Return an MxN data object, where M corresponds to the number of read input channels and N corresponds to the data samples per channel. If you want to, you can also force the user to previously allocate the given data object such that you can get a hint how many samples should be registered.
 - For only one input channel, it is also possible to implement the definition **getVal(QSharedPointer<char> data, QSharedPointer<int> length, ItomSharedSemaphore *waitCond = NULL)** where an allocated char-buffer whose size is defined by *length* is given. Fill in the data samples into the buffer (considering the given length) or use the length value to see how many samples are requested.
- Method **setVal**: This method is called if the user wants the plugin to set data to all selected output channels. The definition is **setVal(const char *data, const int length, ItomSharedSemaphore *waitCond = NULL)**. In case of AD-DA-converter plugins, length is always 1 and *data* must be cast to `ito::DataObject*`. The given data object must then have a size of MxN where M denotes the number of output channels (must correspond to the number of channels to write data) and N is the number of samples. You can then send all samples to each channel either as fast as possible or using a timer, using the timer of the device. This depends on its abilities.

A sample header file of the DataIO's plugin class for AD-converters might look like the following snippet:

```

1  #include "common/addInInterface.h"
2  #include <qsharedpointer.h>
3
4  class MyADConverter : public ito::AddInDataIO
5  {
6      Q_OBJECT
7
8      protected:
9          ~MyADConverter(); /*! < Destructor*/
10         MyADConverter(); /*! < Constructor*/
11
12     public:

```

```

13     friend class MyADConverterInterface;
14     const ito::RetVal showConfDialog(void);    ///! Open the config nonmodal dialog to set cam
15     int hasConfDialog(void) { return 1; }; ///!< indicates that this plugin has got a configur
16
17     private:
18
19     public slots:
20         ito::RetVal getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond);
21         ito::RetVal setParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond);
22
23         ito::RetVal init(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItomSha
24         ito::RetVal close(ItomSharedSemaphore *waitCond);
25
26         ito::RetVal startDevice(ItomSharedSemaphore *waitCond);
27         ito::RetVal stopDevice(ItomSharedSemaphore *waitCond);
28
29         ito::RetVal getVal(void *vpdObj, ItomSharedSemaphore *waitCond);
30         ito::RetVal getVal(QSharedPointer<char> data, QSharedPointer<int> length, ItomSharedSemaph
31         ito::RetVal setVal(const char *data, const int length, ItomSharedSemaphore *waitCond = NU
32
33     private slots:
34         void dockWidgetVisibilityChanged(bool visible);
35 };
36
37 class MyADConverterInterface : public ito::AddInInterfaceBase
38 {
39     Q_OBJECT
40     #if QT_VERSION >= QT_VERSION_CHECK(5, 0, 0)
41     Q_PLUGIN_METADATA(IID "ito.AddInInterfaceBase" )
42     #endif
43     Q_INTERFACES(ito::AddInInterfaceBase)
44     PLUGIN_ITOM_API
45
46     protected:
47
48     public:
49         MyADConverterInterface();
50         ~MyADConverterInterface();
51         ito::RetVal getAddInInst(ito::AddInBase **addInInst);
52
53     private:
54         ito::RetVal closeThisInst(ito::AddInBase **addInInst);
55 };

```

RawIO-Plugins

Further IO plugins are directly inherited from **ito::AddInDataIO**.

Todo

documentation for other IO devices

8.3.12 Plugin class - Actuator

If you want to create a plugin in order to access piezo- or motor-stages, multi-axes-machines,... it is intended to derive your plugin class from **ito::AddInActuator**.

Base idea of any actuator

The actuator interface has been developed with the following base ideas:

- The actuator can consist of different axes. The first axis is indexed with the number 0. Create the read-only, integer parameter **numAxis** for the number of connected axes.
- The plugin contains a set of parameters (like every other plugin), which can be set or get using the public methods **setParam** or **getParam** or the appropriate methods in Python.
- emit the signal **parametersChanged(m_params)** if any parameter has been changed in order to inform the GUI about these changes.
- Some of those parameters must be available, others are optional, but if you implement them, you should follow some rules concerning name and type of the parameter, and of course you can add an infinite list of further parameters (see below).
- Every actuator is executed in its own thread.
- Every actuator can have one configuration dialog and one dockable toolbox, that is directly included in the GUI of **itom**.
- The current position of all axes should be stored in the vector **m_currentPos**, that is a member of class **AddInActuator** (bitmask of enumeration **ito::tActuatorStatus**).
- The new target position for all axes must be stored in the vector **m_targetPos**, that is also a member of class **AddInActuator**.
- The current status of all axes should be stored in the vector **m_currentStatus** (member of class **AddInActuator**).
- Make sure at the initialization of your plugin that all three member vectors are initialized with the size of the numbers of axes.
- Changes to the current status or position are signalled by the signal **actuatorStatusChanged** (class **AddInActuator**). This is usually emitted by calling **sendStatusUpdate(...)**.
- Changes to the target position vector is signalled by the signal **targetChanged** (class **AddInActuator**). This is usually emitted by calling **sendTargetUpdate()**. Connected toolboxes as well as further GUI elements are then informed about the changes.
- Any GUI elements should only get live information about the position and status of the actuator by connecting to these signals (**actuatorStatusChanged** or **targetChanged**), since the communication to GUI elements must be executed across multiple threads.
- Try to only connect to these signals if you really need this information, since the request of live-status and -position-information is time-consuming for certain motors. For example, a dock widget should only connect to these signals, if it is visible. This can be done by overwriting the slot **dockWidgetVisibilityChanged** of class **AddInBase**.
- Methods like **setPosAbs**, **setPosRel**, **setParam**, **calib** or **setOrigin** should only execute their given task if the motor is not moving at this moment. This can be checked using the method **isMotorMoving()**, defined in **AddInActuator** by simply checking the appropriate status flags.
- The method **waitForDone** (pure virtual method of class **AddInActuator**) **has to be overwritten**. This method continuously checks the moving status of all (moving) axes and returns if all requested axes reached their target position, reached a switch or if a time-out occurred. User interrupts are also checked within this function. In case of such an interrupt the axes status are set to *interrupt* as well and the method returns. If the hardware is able to give sophisticated live information about the current status and position of each axis, you can continuously adapt the values in the members **m_currentStatus** or **m_currentPos**; else you can guess these values. Signal changes to these vectors using the methods **sendStatusUpdate()** or **sendStatusUpdate(...)**. If the axes are asynchronously moved, the semaphore **waitCond** has to be released immediately before the loop waiting for the target position starts. Then the caller can directly continue working. In synchronous mode (default behaviour), **waitCond** is only released, if no requested axis is moving any more.

- There is a slot **requestStatusAndPosition(bool sendActPosition, bool sendTargetPos)**, defined in **ito::AddInActuator**. This slot is invoked e.g. by a toolbox or configuration dialog in order to force the actuator to directly emit the signals **actuatorStatusChanged** and/or **targetChanged**. The original caller is then immediately informed about the current status and position values. Overload this function if you want to update **m_currentStatus**, **m_currentPos** or **m_targetPos** before they are emitted to the caller. In the default implementation, they are emitted as they are.

Programming steps

In order to program the actuator plugin, follow these steps:

1. Create the header and source file for your plugin “MyActuatorPlugin”.
2. Create the interface (or factory) class “MyActuatorPluginInterface”. For details about how to create such an interface class, see *Plugin interface class*.
3. Create the plugin class “MyActuatorPlugin” with respect to the exemplary implementation, given in the next section.
 - Consider which internal parameters, that can be read and/or written by the user, your plugin has. Add these parameters in the constructor of your plugin to the **m_params**-vector.
 - Implement the **init**-method that gets the initial parameters, defined in the interface class.
 - Implement the methods **getParam** and **setParam**, which are the getter- and setter-methods for the internal parameters.
 - Implement the motor-specific methods, including **waitForDone**

Actuator plugin class

A sample header file of the actuator’s plugin class is illustrated in the following code snippet:

```
1  #define ITOM_IMPORT_API
2  #define ITOM_IMPORT_PLUGIN_API
3
4  #include "../common/addInInterface.h"
5
6  #include "dialogMyMotor.h"
7  #include "dockWidgetMyMotor.h"
8
9  class MyMotor : public ito::AddInActuator
10 {
11     Q_OBJECT
12
13     protected:
14         ~MyMotor() {};          /*! < Destructor*/
15         MyMotor(); /*! < Constructor*/
16
17         ito::RetVal waitForDone(int timeoutMS = -1, QVector<int> axis = QVector<int>()) /*if empty
18
19     public:
20         friend class MyMotorInterface;
21         const ito::RetVal showConfDialog(void);          /*!< Opens the modal configuration dialog (ca
22         int hasConfDialog(void) { return 1; }; /*!< indicates that this plugin has got a configur
23
24     public slots:
25         /// get/set parameters
26         ito::RetVal getParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond = NULL,
27         ito::RetVal setParam(QSharedPointer<ito::Param> val, ItomSharedSemaphore *waitCond = NULL,
28
29         /// init/close method
30         ito::RetVal init(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItomSha
```

```

31         ito::RetVal close(ItomSharedSemaphore *waitCond);
32
33         ///! calibration for single or multiple axis
34         ito::RetVal calib(const int axis, ItomSharedSemaphore *waitCond = NULL);
35         ito::RetVal calib(const QVector<int> axis, ItomSharedSemaphore *waitCond = NULL);
36
37         ///! current axis position is new zero-position
38         ito::RetVal setOrigin(const int axis, ItomSharedSemaphore *waitCond = NULL);
39         ito::RetVal setOrigin(const QVector<int> axis, ItomSharedSemaphore *waitCond = NULL);
40
41         ///! Reads out status request answer and gives back ito::retOk or ito::retError
42         ito::RetVal getStatus(QSharedPointer<QVector<int> > status, ItomSharedSemaphore *waitCond);
43
44         ///! get current position of single or multiple axis (in mm or degree)
45         ito::RetVal getPos(const int axis, QSharedPointer<double> pos, ItomSharedSemaphore *waitCond);
46         ito::RetVal getPos(const QVector<int> axis, QSharedPointer<QVector<double> > pos, ItomSharedSemaphore *waitCond);
47
48         ///! move one or more axis to certain absolute positions (in mm or degree)
49         ito::RetVal setPosAbs(const int axis, const double pos, ItomSharedSemaphore *waitCond = NULL);
50         ito::RetVal setPosAbs(const QVector<int> axis, QVector<double> pos, ItomSharedSemaphore *waitCond = NULL);
51
52         ///! move one or more axis by certain relative distances (in mm or degree)
53         ito::RetVal setPosRel(const int axis, const double pos, ItomSharedSemaphore *waitCond = NULL);
54         ito::RetVal setPosRel(const QVector<int> axis, QVector<double> pos, ItomSharedSemaphore *waitCond = NULL);
55
56         ///! if this slot is triggered, the current status and position is emitted (e.g. for actual position)
57         ito::RetVal RequestStatusAndPosition(bool sendActPosition, bool sendTargetPos);
58
59         ///! ito::RetVal requestStatusAndPosition(bool sendCurrentPos, bool sendTargetPos); //!see note
60
61     private slots:
62         void dockWidgetVisibilityChanged( bool visible ); /*!< this slot is invoked if the visibility changes */
63 };

```

The corresponding source file should start with something like this:

```

1  #define ITOM_IMPORT_API
2  #define ITOM_IMPORT_PLOTAPI
3
4  #include "yourHeaderFile.h"
5
6  //implement your code here

```

Signalling the current position and status of any axes

Each actuator has the possibility to signalize the target position, the current position and the current status of each axis. Then its own toolbox or other widgets (general: listeners) can be connected to the corresponding signals in order to be informed about the current activity. The base class **ito::AddInActuator** provides the necessary structures for this:

1. The vector **m_currentPos** must be initialized to a length corresponding to the number of axes and contains the current position of every axis using the units stated below. Whenever the actuator registers a change of any current position, the corresponding value should be changed as well. Listeners are finally informed about this change by calling the method

```
sendStatusUpdate(false)
```

The argument **false** means that not only a change of the current status happened, but also a change of any current position. This method internally emits the signal **actuatorStatusChanged**.

2. The vector **m_targetPos** must also be initialized to a length corresponding to the number of axes. Whenever a positioning operation starts, set the target value of specific axes to the new target value and call

```
sendTargetUpdate()
```

that finally emits the signal **targetChanged**.

3. The status of every axis is stored in the vector **m_currentStatus**. Each item in this vector with a length corresponding to the number of axes, contains an OR combination of the enumeration **ito::tActuatorStatus**. Whenever the status of any axis changes, change its status value, too and use **sendStatusUpdate(true/false)** in order to also emit the signal **actuatorStatusChanged**.

The enumeration **ito::tActuatorStatus** contains the following values that are grouped by specific mask values:

The **moving flags** contain flags about the current moving status of any axis (bits containing to this group are contained in the mask **ito::actMovingMask**):

- **ito::actuatorUnknown**: The current status of this axis is unknown
- **ito::actuatorInterrupted**: The movement of this axis has been interrupted and no further commands followed
- **ito::actuatorMoving**: The axis is currently moving (or is supposed to move)
- **ito::actuatorAtTarget**: The axis reached its target position (this is the default value)
- **ito::actuatorTimeout**: A timeout occurred during the movement of this axis

The **status flags** inform about the general status of any axis (bits containing to this group are set in the mask **ito::actStatusMask**):

- **ito::actuatorAvailable**: This axis is available (usually set)
- **ito::actuatorEnabled**: This axis is enabled and can be driven (usually set, but there are drivers that allowing disabling selected axis)

Axes that have got any reference or end switches can signal related status information using the **switches flags**. All bits belonging to this group are set in the mask **ito::actSwitchesMask** divided into **ito::actEndSwitchMask** and **ito::actRefSwitchMask**):

- **ito::actuatorEndSwitch**: This bit is set if any (unknown) end switch was reached
- **ito::actuatorLeftEndSwitch**: This bit is additionally set if the left end switch was reached
- **ito::actuatorRightEndSwitch**: This bit is additionally set if the right end switch was reached
- **ito::actuatorRefSwitch**: This bit is set if any (unknown) reference switch was reached
- **ito::actuatorLeftRefSwitch**: This bit is additionally set if the left reference switch was reached
- **ito::actuatorRightRefSwitch**: This bit is additionally set if the right reference switch was reached

You can either manually set the necessary bit-combination of moving, status and switch flags for signalling the right status of the axis. There are three methods defined in **ito::AddInActuator** that simplify this process:

```
setStatus(int &status, const int newFlags, const int keepMask = 0)
setStatus(const QVector<int> &axis, const int newFlags, const int keepMask = 0)
```

Use this methods to set the status of one or multiple axis. The parameter **newFlags** should contain an or-combination of all flags that should be set. The status flags are then set to this value (hence, old values are overwritten). If you want to keep the current bit values of a certain group, pass the specific mask as argument **keepMask**. For instance, if you want the status of the second axis to **actuatorMoving** without changing the **status flags**, use the following command:

```
setStatus(m_currentStatus[1], ito::actuatorMoving, ito::actStatusFlags)
# this command will set all bits of the switches mask to 0!
```

The equivalent command for multiple axis, requires a vector with axes-indices as first argument. This example does the same for the first and third axis:

```
QVector<int> axis;
axis << 0 << 2;
setStatus(axis, ito::actuatorMoving, ito::actStatusFlags)
```

The similar commands **replaceStatus**

```
replaceStatus(int &status, const int existingFlag, const int replaceFlag)
replaceStatus(const QVector<int> &axis, const int existingFlag, const int replaceFlag)
```

can be used to replace one status flag by another one without changing the other bits. If the bit corresponding to the **existingFlag** is set, it is set to zero and the bit of the **replaceFlag** is set to 1. In the following example, the flag of the first axis is set from moving to atTarget:

```
replaceStatus(m_currentStatus[0], ito::actuatorMoving, ito::actuatorAtTarget)
```

After using one of these functions to set the current status, call **sendStatusUpdate** to emit the signal **actuatorStatusChanged** such that connected listeners can for instance visualize the current status.

Interruption of movement

It is possible to implement an interrupt button in the toolbox of the actuator that becomes active once at least one axis is moving. Once the button is clicked it must **directly** call the thread-safe function **setInterrupt()** of the actuator plugin (If the toolbox inherits from **ito::AbstractAddInDockWidget** call its method **setActuatorInterrupt()**).

In the method **waitForDone** regularly check if the interrupt flag has been set, using the actuator's method **isInterrupted()**. If this method returns true, set the moving state of all moving axes to **ito::actuatorInterrupted** and return with an appropriate return value, like:

```
return ito::RetVal(ito::retError, 0, "movement interrupted");
```

Note: Once **isInterrupted()** returns true, the internal interrupt flag is reset to false. Therefore consider to call this function to reset the interrupt flag if desired (e.g. at the begin of the next movement).

Parameters and Unit Conventions

In order to have a unified behaviour of all actuator plugins, respect the following unit conventions. That means, the plugin should store related parameters using these conventions, such that **getParam** and **setParam** returns and obtains values using these units. Internally, it is sometimes necessary to convert these units to the units required by the interface of the real actuator device.

- Length values in **mm**
- Angles in **degree**
- Velocity in **mm/sec** or **degree/sec**
- Acceleration, deceleration in **mm/sec^2** or **degree/sec^2**

Implement the following mandatory parameters in the map **m_params**:

- **“name”**: {string | readonly} name of the plugin
- **“numaxis”**: {int | readonly} number of connected axes
- **“async”**: {int, [0,1]} If 1: asynchronous movement. Methods like **setPosAbs** or **setPosRel** only start the movement and immediately return. Hence, the *waitCond* in **waitForDone** is directly released before the loop waiting for the end of the movement is executed. If 0: synchronous movement (default). **setPosAbs** and **setPosRel** block until the end of the movement, hence, *waitCond* in **waitForDone** is only released at the end of the movement. Since **waitForDone** always is running during the movement, the plugin thread is blocked and no further commands can be executed, even in asynchronous mode.

If desired implement the following optional parameters in the map **m_params**:

- **“speed”**: {double or doubleArray} Desired speed for the axes. If *double*, the speed holds for all axes, else the *doubleArray* must have the same length than the number of axes, holding the axis specific speed values. Make sure, that it is not possible to set an array of another length in **setParam**.

- “**accel**”: {double or doubleArray} Acceleration values (similar to *speed*)
- “**decel**”: {double or doubleArray} Deceleration values (similar to *speed*)

8.3.13 Plugin class - Algo

An **algorithm plugin** can provide an arbitrary number of filter-methods and widgets, hence external windows, dialogs... which can be displayed by **itom**.

Filter-Method A filter-method is any algorithm, which is parameterized with a set of mandatory and optional parameters and may return a set of output parameters. Similar to the flexible initialization of plugins of type **dataIO** or **actuator**, the default-values of these parameters are given by the plugin itself and can also be different for each filter-method. Please consider that a filter-method can finally be called from different threads, therefore it is not allowed to directly use any GUI-elements in such plugins. If the appropriate method is called by the python function **filter**, it is executed in the python thread, if the method is called from any toolbox or dialog, it is usually called in their context (main thread) but it can also be, that the filter-method is executed by any other worker thread. If a filter-method only consists of ordinary algorithmic components, you should not get any problems with these kind of flexibility.

Widgets The widget-methods of the plugin will be called in the same way than filter-methods with a set of mandatory and optional parameters. Then, the widget-method creates a new instance of a widget, window or any other GUI element, derived from **QWidget** is returned and can then be displayed. This method will always be called in the context of the main thread (GUI-thread).

Plugin-Structure

Like all the other types of plugins, the **Algo**-plugin must at least consist of two classes. The first (*here*: **MyAlgoPluginInterface**) is the interface, or factory class, necessary for the successful load of the plugin by the **Qt**-plugin system. The “real” plugin class (*here*: **MyAlgoPlugin**) is then accessed by **itom** through appropriate methods in class **MyAlgoPluginInterface**.

Factory-Class

The structure and exemplary implementation of a factory class **MyAlgoPluginInterface** is mainly explained in *Plugin interface class*. Since the main class **MyAlgoPlugin** is mainly behaving like a static, singleton class, it will never be instantiated by the user, like it is the case for plugins of type **dataIO** or **actuator**. Therefore it makes no sense to define some default mandatory and optional parameters for the constructor of **MyAlgoPlugin**, such that the programmer should the vectors **m_initParamsMand** and **m_initParamsOpt** be unchanged.

Plugin-Class

The raw scheme for your plugin-class is as follows:

Header-File (myPluginAlgo.h)

```
1  #ifndef MYALGOPLUGIN_H
2  #define MYALGOPLUGIN_H
3
4  #include "../common/addInInterface.h" //or similar path
5
6  class MyPluginAlgoInterface : public ito::AddInInterfaceBase
7  {
8  ... //see in documentation for interface-class
9  };
10
11 class MyPluginAlgo : public ito::AddInAlgo
12 {
13     Q_OBJECT
```



```

14
15 protected:
16     MyPluginAlgo(int uniqueID);
17     ~MyPluginAlgo() {};
18
19 public:
20     friend class MyAlgoPluginInterface;
21
22     //filter-method1
23     static ito::RetVal filter1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt, ItoSharedSemaphore *waitCond);
24     static ito::RetVal filter1Params(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItoSharedSemaphore *waitCond);
25
26     //for every further filter-method define another pair like above
27
28     //widget-method1
29     static QWidget* widget1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt, ItoSharedSemaphore *waitCond);
30     static ito::RetVal widget1Params(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItoSharedSemaphore *waitCond);
31
32     //for every futher widget-method define another pair like above
33
34 public slots:
35     ito::RetVal init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt, ItoSharedSemaphore *waitCond);
36     ito::RetVal close(ItoSharedSemaphore *waitCond);
37 };
38
39 #endif

```

Source File (myPluginAlgo.cpp)

```

1 #define ITOM_IMPORT_API
2 #define ITOM_IMPORT_PLOTAPI
3
4 #include "myPluginAlgo.h"
5
6 //implement your code here

```

First of all, our algorithm plugin class is derived from the class **AddInAlgo** from within the **ito**-namespace. This base class is defined in the *addInInterface.h* header that has to be included. Again, our plugin is ultimately derived from **QObject** and the *Q_OBJECT* macro must appear in the class definition in order to be able to use any services provided by Qt's meta-object system, such as the signal slot mechanisms. Additionally our plugin class has to be a friend of its interface class (see section *Plugin interface class*), such that the factory (interface) class is allowed to access the protected constructor.

Both, the pair of methods for one filter-method and one widget-method consists of the real filter- or widget-method (*here*: named with *filter1* and *widget1*) and their corresponding parameter-methods which generates the default vectors for the mandatory, optional and output (filters only) parameters. This method is necessary, since the **itom** base application and the python scripts have no way of knowing what parameters the filter methods or widget generation methods expect.

These default-parameter methods have the following implementation:

```

1 ito::RetVal MyAlgoPlugin::filter1Params(QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt, ItoSharedSemaphore *waitCond)
2 {
3     ito::Param param;
4     ito::RetVal retval = ito::retOk;
5     retval += prepareParamVectors(paramsMand, paramsOpt, paramsOut);
6     if(retval.containsError()) return retval;
7
8     param = ito::Param("mand1", ito::ParamBase::DObjPtr | ito::ParamBase::In, NULL, tr("description of mandatory parameter 1"));
9     paramsMand->append(param);
10    param = ito::Param("mand2", ito::ParamBase::String | ito::ParamBase::In, NULL, tr("description of mandatory parameter 2"));
11    paramsMand->append(param);
12    param = ito::Param("opt1", ito::ParamBase::Double | ito::ParamBase::In, 0.0, 1.0, 0.0, tr("description of optional parameter 1"));
13    paramsOpt->append(param);

```

14
15
16
17
18
19
20
21

```

    param = ito::Param("return1", ito::ParamBase::Int | ito::ParamBase::Out, NULL, tr("description
    paramsOut->append(param);
    param = ito::Param("return2", ito::ParamBase::String | ito::ParamBase::Out, NULL, tr("descripti
    paramsOut->append(param);

    return retval;
}

```

In this exemplary case, the method **filter1** requires two mandatory parameters, where the first one is a dataObject-pointer and the second one is a string. Additionally one can give one optional parameter, namely a double-value. This filter does not have output-parameters.

Note: Please don't forget to specify your parameters with the flags **In**, **Out** or **In!Out**. Mandatory and optional parameters can not be **Out** only, however output parameters must be **Out** only. Please consider that pure **Out** values is only allowed for non-pointer-type values (hence: Char, Int, Double, String, IntArray, CharArray and DoubleArray are allowed). For widget-methods, the output-value default implementation is ignored and therefore the corresponding vector should not be changed.

In line 3 of the method above, the three arguments are checked for not being NULL and are cleared within the method **prepareParamVectors**, which is already defined in class **ito::AddInAlgo**. Next, the default implementations of parameters (see *Parameter-Container class of itom*) are created and appended to the corresponding vector. Finally the method is called in order to get the three default vectors. Then, the values are merged with the values given by the user or other plugins and the real filter- or widget-method is called with the same vectors, however each parameter is now casted from **Param** to **ParamBase**.

Note: In order to have an efficient call, the parameter method is only called once at startup of **itom** and the default implementations are hashed by **ito::AddInManager**. The consequence for the plugin programmer is, that it is not allowed to have time- or situation-dependent changes in the default-values. This might not be considered. Only the implementation at startup is relevant and must not be changed!

The *plugin*-methods give the user additionally the possibility to get a readable output of the set of desired parameters including their descriptions and types in the command line by using the following python-command:

```
filterHelp("MyAlgoPlugin")
```

If the argument string does not fit to any specific filter, an enumeration of all filters containing this string will be printed.

Now the filter-method or widget-method itself can be implemented:

Filter-Methods

After that you implemented the parameter-method in order to generate default parameters for your filter-method (see section above), you can now implement the filter-method itself. The implementation might follow this scheme:

1
2
3
4
5
6
7
8
9
10
11
12
13

```

ito::RetVal MyAlgoPlugin::filter1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *p
{
    ito::RetVal retval = ito::retOk;

    //1. Section. Getting typed in or in/out parameters from paramsMand and paramsOpt
    // Make sure that you access only parameters, that have been defined in the corresponding pa
    // The order and type is important.

    //possibility 1 (index-based access):
    const ito::DataObject *dObj = (*paramsMand)[0].getVal<const ito::DataObject*>();
    const char *filename = (*paramsMand)[1].getVal<char*>(); //don't delete this pointer (borrowed
    double opt1 = (*paramsOpt)[0].getVal<double>();
}

```

```

14 //possibility 2 (name-based access):
15 const ito::DataObject *dObj2 = (const ito::DataObject*)ito::getParamByName(paramsMand, "mand2");
16 const char *filename2 = ito::getParamByName(paramsMand, "mand2", &retval)->getVal<char*>();
17 double opt2 = ito::getParamByName(paramsOpt, "opt1", &retval)->getVal<double>();
18
19 //2. Section. Algorithm.
20 // include here your algorithm. make sure, that you only change values of pointer-based parameters
21 // that you have defined with the flags In|Out.
22
23 //3. Section. Optionally put results into the paramsOut-vector
24 // Make sure that you defined the corresponding parameter with right type in the corresponding method.
25 // The implementation in the next lines is only one example and has not be defined before. But you can
26 (*paramsOut)[0].setVal<int>(2);
27 (*paramsOut)[1].setVal<char*>("we are done");
28
29 return retval;
30 }

```

For the parsing of the given input parameters, you can (like stated above) either directly access them using their index in the vector or you can use the high-level method **getParamByName**, which is provided in the file **helperCommon.h** and **helperCommon.cpp** in the **common**-folder. If you want to use this method, integrate both files in your project and include the header file in your plugin file.

Todo

Move `getParamByName` into the API and remove it from `helperCommon`. Correct the documentation.

Note: Always make sure, that if you are access (read or write) any parameter in any of the three vectors, you must have the specific parameter defined in the corresponding parameter method. If you defined there a parameter of certain type and appended it to one of the vectors, you can be sure, that a parameter of same type is available at the same position in the arguments of the filter-method call.

Note: If you want to use methods provided by the **itom**-API, see *itom API* and consider the additional lines of code in your implementation.

Widget-Method (GUI-Extensions)

If you want to provide a user-defined window, dialog or widget (which is then rendered into a dialog), you have to implement an appropriate method which follows this base structure:

```

1 QWidget* MyAlgoPlugin::widget1(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt)
2 {
3     //1. Section. Getting typed in or in/out parameters from paramsMand and paramsOpt
4     // Make sure that you access only parameters, that have been defined in the corresponding parameter method.
5     // The order and type is important. Do it like in the method 'filter1' above.
6
7     //2. Pre-requisite: You have in your plugin project a class, which is derived from QMainWindow
8     // This class can also include an ui-file, which has been designed using the QtDesigner. The ui-file
9     // must be included in the plugin project.
10
11     //Create an instance of that class and return it. The instance is deleted by the caller of the method.
12     Widget1 *win = new Widget1( /* your parameters */ );
13     QWidget *widget = qobject_cast<QWidget*>(win); //cast it to QWidget, if it isn't already.
14     if(widget == NULL)
15     {
16         retval += ito::RetVal(ito::retError,0,tr("The widget could not be loaded").toLatin1().data());
17     }
18     return widget; //NULL in case of error
19 }

```

The widget will then be shown if the user created it by the GUI or the widget is wrapped by an instance of the python class **ui** (part of module **itom**). Then you can interact with elements of the widget using python like you can do it with every user interface created with **QtDesigner**. You can also connect some python methods with signals of your widget or call slots of your widget. This is also the same behaviour like dialogs have, which only have been created with **QtDesigner** and then loaded by an instance of class **ui** using python.

Publish Filter- and Widget-Methods at Initialization

The most important step in the development of an algorithm plugin is to publish all created filter- and widget-methods. By that process, the methods will be made available to **itom**, such that they can be used by the python scripting language, the GUI or other plugins. The publishing is done in the method **init** of your plugin. A exemplary implementation is as follows:

```

1  ito::RetVal MyAlgoPlugin::init(QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *param
2  {
3      ItomSharedSemaphoreLocker locker(waitCond);
4
5      ito::RetVal retval = ito::retOk;
6      FilterDef *filter = NULL;
7      AlgoWidgetDef *widget = NULL;
8
9      //publish your filter-methods here, example:
10     filter = new FilterDef(WLIfilter, WLIfilterParams, tr("description").toLatin1().data(), ito::
11     m_filterList.insert("filterName", filter);
12
13
14     //publish your dialogs, main-windows, widgets... here, example:
15     widget = new AlgoWidgetDef(widget1, widget1Params, tr("description").toLatin1().data(), ito::
16     m_algoWidgetList.insert("widgetName", widget);
17
18     if (waitCond)
19     {
20         waitCond->returnValue = retval;
21         waitCond->release();
22     }
23
24     return retval;
25 }
```

For registering filter- and widget-methods, you have to create a new instance of the classes **FilterDef** or **AlgoWidgetDef** respectively and insert these newly created instances to the maps **m_filterList** or **m_algoWidgetList** respectively. Their name in the map finally is the name of the filter or widget and must be unique within **itom**; else the filter-method or widget-method can not be loaded at startup of **itom**.

The constructor of class **FilterDef** has the following arguments:

- *AddInAlgo::t_filter* **filterFunc** is the pointer to the static filter-method (*here*: filter1)
- *AddInAlgo::t_filterParam* **filterParamFunc** is the pointer to the corresponding parameter method (*here*: filter1Params)
- *QString* **description** is a description string of the filter
- *ito::AddInAlgo::tAlgoCategory* **category** is an optional value of the category enumeration, the filter belongs too (*default*: *ito::AddInAlgo::catNone*)
- *ito::AddInAlgo::tAlgoInterface* **interf** is an optional value of the interface enumeration, the filter fits to (*default*: *ito::AddInAlgo::iNotSpecified*)
- *QString* **interfaceMeta** is depending on the chosen interface an additional meta string (*default*: *QString()*)

For a full reference of class **FilterDef**, see [FilterDef](#).

The constructor of class **AlgoWidgetDef** has the following arguments:

- `AddInAlgo::t_algoWidget` **algoWidgetPFunc** is the pointer to the static widget-method (*here*: `widget1`)
- `AddInAlgo::t_filterParam` **algoWidgetParamFunc** is the pointer to the corresponding parameter method (*here*: `widget1Params`)
- `QString` **description** is a description string of the widget or GUI-element.
- `ito::AddInAlgo::tAlgoCategory` **category** is an optional value of the category enumeration, the widget-method belongs too (*default*: `ito::AddInAlgo::catNone`)
- `ito::AddInAlgo::tAlgoInterface` **interf** is an optional value of the interface enumeration, the widget-method fits to (*default*: `ito::AddInAlgo::iNotSpecified`)
- `QString` **interfaceMeta** is depending on the chosen interface an additional meta string (*default*: `QString()`)

For a full reference of class **AlgoWidgetDef**, see [AlgoWidgetDef](#).

For information about available algorithm categories and interfaces, see

Algorithm Categories and Interfaces

You can assign to every filter- or widget-method a certain category and/or algorithm interface, the method belongs or fits to. This is useful such that **itom** can ...

- categorize your filter or widget.
- is able to integrate the filter in specific parts of the GUI, e.g. the file load or save process or dialog.
- might use your filters in image processing chains.
- ...

Algorithm Categories

Algorithm Interfaces A programmer who wants to implement a filter or widget-method that fits to a specific algorithm interface can be forced by the interface to consider a certain set of rules. These rules can be:

- The first **n** mandatory parameters can be determined. For each of these parameters the type and optionally some constraints by adding a meta information instance to the parameter are given.
- The first **m** output parameter can be determined. Type and meta information (optional) are given as well).
- The maximum number of mandatory parameters is determined, but can also be set to infinity (`INT_MAX`).
- The maximum number of optional parameters is determined, but can also be set to infinity (`INT_MAX`).
- The maximum number of output parameters is determined, but can also be set to infinity (`INT_MAX`).
- The syntax or general form of the meta information string, added to the classes **FilterDef** or **AlgoWidgetDef** can be given.

If a filter or widget-method pretends to implement a certain interface, **itom** checks at startup if the constraints are fulfilled or if this is not the case, the filter or widget-method is rejected and cannot be used within **itom**. The reason for the rejection can be seen by open the dialog **loaded plugins...** within the **help**-menu of **itom**.

The following interfaces are available (in the enumeration **tAlgoInterface**, member of class `ito::AddInAlgo`:

iNotSpecified This is the default implementation and indicates that your filter or widget-method does not fit to any interface.

iReadDataObject Filters fitting to this interface provide the functionality to read a certain file and load its content to an instance of dataObject.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 dataObject	DObjPtr, In, Out	DataObject, where the file is loaded to	None
#2 filename	String, In	absolute filename of the file to load	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iWriteDataObject Filters fitting to this interface provide the functionality to export a dataObject with a specific file format.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 dataObject	DObjPtr, In	DataObject that should be exported	None
#2 filename	String, In	absolute filename of the file	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iReadPointCloud Filters fitting to this interface provide the functionality to read a certain file and load its content to an instance of pointCloud.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 pointCloud	PointCloudPtr, In, Out	PointCloud, where the file is loaded to	None
#2 filename	String, In	absolute filename of the file to load	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iWritePointCloud Filters fitting to this interface provide the functionality to export a pointCloud with a specific file format.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 dataObject	PointCloudPtr, In	PointCloud that should be exported	None
#2 filename	String, In	absolute filename of the file	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iReadPolygonMesh Filters fitting to this interface provide the functionality to read a certain file and load its content to an instance of polygonMesh.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 pointCloud	PolygonMeshPtr, In, Out	PolygonMesh, where the file is loaded to	None
#2 filename	String, In	absolute filename of the file to load	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

iWritePolygonMesh Filters fitting to this interface provide the functionality to export a polygonMesh with a specific file format.

Argument limitations

Parameter group	Max. number of parameters
mandatory parameters	infinity
optional parameters	infinity
output parameters	0

Mandatory parameters

Parameters	Type	Description	Further limitations (meta information)
#1 polygonMesh	PolygonMeshPtr, In	PolygonMesh that should be exported	None
#2 filename	String, In	absolute filename of the file	None

Output parameters

- No -

Meta information string

This string must contain the file-filters with file-endings that this filter is able to load. Different file filters are separated by a double-semicolon (;). Each filter begins with its name (arbitrary string), followed by a space and a sequence of file-endings within a pair of brackets. Examples are:

- Images (*.bmp *.png *.jpg *.pgm)
- Bitmap (*.bmp);;JPEG (*.jpg)
- Text-Files (*.txt)

Definition of new algorithm interfaces At first the categories and interfaces are determined as an enumeration value in the enumerations `ito::AddInAlgo::tAlgoCategory` and `ito::AddInAlgo::tAlgoInterface` (file `addInInterface.h`).

The constraints and rules for every interface are implemented in the methods `init` and `getTags` of class `AlgoInterfaceValidator`. Besides the syntax of the meta information string all necessary information is given in the method `init`.

The method-types `t_filter`, `t_algoWidget` and `t_filterParam` are defined by the following *typedef*:

```
typedef ito::RetVal (* t_filter) (QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt);
typedef QWidget* (* t_algoWidget) (QVector<ito::ParamBase> *paramsMand, QVector<ito::ParamBase> *paramsOpt);
typedef ito::RetVal (* t_filterParam) (QVector<ito::Param> *paramsMand, QVector<ito::Param> *paramsOpt);
```

Finish and close plugin

Finally, the `close` method can be implemented, those structure is usually unchanged:

```
1 ito::RetVal MyAlgoPlugin::close (ItomSharedSemaphore *waitCond)
2 {
3     ItomSharedSemaphoreLocker locker (waitCond);
4     ito::RetVal retval = ito::retOk;
5
6     if (waitCond)
7     {
8         waitCond->returnValue = retval;
9         waitCond->release();
10    }
11
12    return retval;
13 }
```

The `close`-method does nothing but immediately releasing the parameter `waitCond`. For more information about `ItomSharedSemaphore`, see [ItomSharedSemaphore](#).

FilterDef

class `ito::AddInAlgo::FilterDef`
container for publishing filters provided by any plugin

Public Functions

`FilterDef()`
< empty, default constructor
constructor with all necessary arguments.

Public Members

`t_filter m_filterFunc`
function pointer (unbounded, static) for filter-method

`t_filterParam m_paramFunc`
function pointer (unbounded, static) for filter's default parameter method

`ito::AddInInterfaceBase * m_pBasePlugin`
interface (factory) instance of this plugin (will be automatically filled)

`QString m_name`
name of filter

QString **m_description**
description of filter

ito::AddInAlgo::tAlgoCategory **m_category**
category, filter belongs to (default: catNone)

ito::AddInAlgo::tAlgoInterface **m_interface**
algorithm interface, filter fits to (default: iNotSpecified)

QString **m_interfaceMeta**
meta information if required by algorithm interface

AlgoWidgetDef

class **ito::AddInAlgo::AlgoWidgetDef**
container for publishing widgets provided by any plugin

Public Functions

AlgoWidgetDef()
< empty, default constructor
constructor with all necessary arguments.

virtual **~AlgoWidgetDef()**
destructor

Public Members

t_algoWidget **m_widgetFunc**
function pointer (unbounded, static) for widget-method

t_filterParam **m_paramFunc**
function pointer (unbounded, static) for widget's default parameter method

ito::AddInInterfaceBase * **m_pBasePlugin**
interface (factory) instance of this plugin (will be automatically filled)

QString **m_name**
name of widget

QString **m_description**
description of widget

ito::AddInAlgo::tAlgoCategory **m_category**
category, widget belongs to (default: catNone)

ito::AddInAlgo::tAlgoInterface **m_interface**
algorithm interface, widget fits to (default: iNotSpecified)

QString **m_interfaceMeta**
meta information if required by algorithm interface

8.3.14 Create a new plugin via CMake

Plugins for **itom** are also created using **CMake**. Therefore the sources and the project files of the plugins will also be separated, like it is also the case for **itom**, the designer plugins and all other plugins. You can either put your source files in any subfolder of the location where all sources of your plugins lie or you can put it at any arbitrary location on your harddrive.

The source folder of your plugin mainly consists of these files:

1. **CMakeLists.txt**. This is the project file of your plugin, where you insert the files that are included in your plugin or libraries, the plugin should link with. This file is finally interpreted by **CMake** in order to create the real project files, adapted to your generator (e.g. Visual Studio).
2. **yourPlugin.h** and **yourPlugin.cpp**. This are the main header and source files of your plugin.
3. **dialogYourPlugin.h** and **dialogYourPlugin.cpp** (optional, not for algorithms). Use these files if you want to provide a configuration dialog for your plugin (can also be added later).
4. **dockWidgetYourPlugin.h** and **dockWidgetYourPlugin.cpp** (optional, not for algorithms). Use these files if you want to provide a dock widget of your plugin that is inserted into **itom**'s main window (can also be added later).
5. **pluginVersion.h** (optional). This header contains some defines for your current plugin version. It should be included in **yourPlugin.cpp** and under MSVC / windows in **version.rc**.
6. **version.rc** (optional, only under MSVC / windows). Under windows the content of this file will be automatically added to the meta-Data of your DLL.

Templates

You will find templates for the most important files needed for creating a new plugin in the folder **pluginTemplates** of the SDK-directory of **itom**.

Copy the file **CMakeLists.txt** from this template-directory and copy it to the source directory of your plugin. Open this file with an arbitrary editor. The commands, starting with #, give you hints where you need to adapt this file. Further details about the syntax of these files can be found under <http://www.cmake.org/cmake/help/documentation.html>.

Furthermore, you will find template implementations of an actuator, algorithm and camera plugin in the **plugin-Templates** folder. Copy the corresponding header and source file of your type into your source directory and start replacing the template strings by your versions, e.g. you need to change the name of your plugin hence the class names.

Generate project

If you placed the source files of your plugin into a subdirectory of an existing multi-plugin project, then you need to add the subfolder of your plugin into the file **CMakeLists.txt** of the parent-folder. This file usually already contains a lot of subdirectories, added by the CMake-command **ADD_SUBDIRECTORY** and has the following form:

```
project(itom_plugins) #name of the overall project

cmake_minimum_required(VERSION 2.8)

OPTION(BUILD_UNICODE "Build with unicode charset if set to ON, else multibyte charset." ON)
OPTION(BUILD_SHARED_LIBS "Build shared library." ON)
OPTION(BUILD_TARGET64 "Build for 64 bit target if set to ON or 32 bit if set to OFF." OFF)
SET(ITOM_DIR "" CACHE PATH "base path to itom")

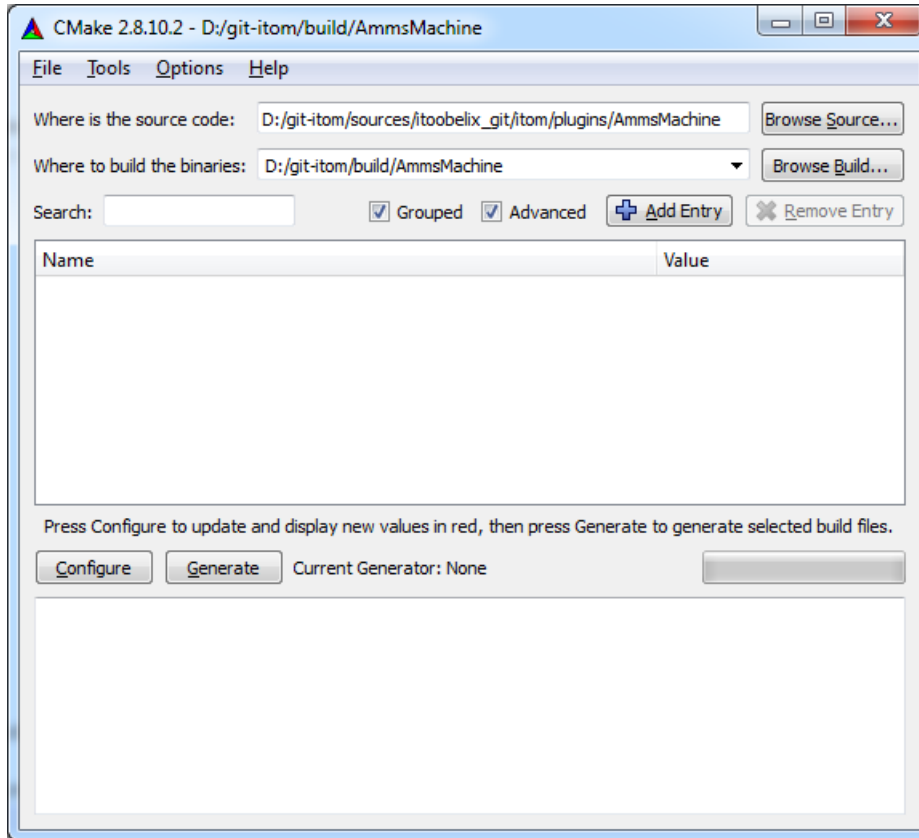
# Insert the following section for your plugin

# by this option, the plugin can be unchecked in order to
# avoid its generation in cmake.
OPTION(PLUGIN_YOURNAME "Build with this plugin." ON)
if (PLUGIN_YOURNAME)
    ADD_SUBDIRECTORY(YOURSUBDIRECTORY)
endif (PLUGIN_YOURNAME)
```

Then, reconfigure and regenerate the overall plugin project or simply run the project **ZERO_CHECK** of your overall plugin solution such that the new plugin is generated and added to the overall solution. If you have no

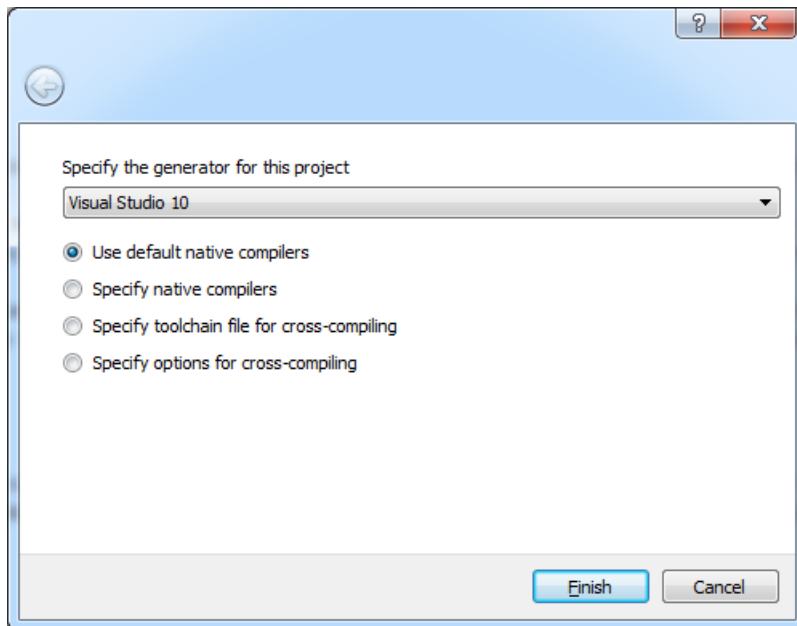
idea about configuring and generating a **CMake** project, continue reading or check the big [CMake-section](#) about generating **itom** itself.

In order to generate your plugin as single solution, open the **CMake GUI** and choose the source directory of your plugin as source directory and any arbitrary folder as build directory. The solution and project files are then generated in this build directory.



Then click the configure button to start the configuration. At first, you will be asked for a generator. See [Build with CMake](#) for more information about generators.

Then, another configuration process is started. Usually, it is now necessary to set the variable **ITOM_SDK_DIR** to the directory of **itom**'s SDK, usually located in **itom**'s build directory. Then click **configure** again.



Next, you probably need to indicate the location of the build-directory of **OpenCV** on your harddrive (variable **OpenCV_DIR** in the group **OpenCV** or **Ungrouped Entries** if you activated the **Grouped** checkbox).

Continue clicking **configure** until there are no more warnings. Then you can press **Generate** in order to build the project files. Change then to the build-directory and open the recently build solution file. In you afterwards change the **CMakeLists.txt**-file, you don't need to explicitly run the CMake-GUI again, since Visual Studio is also able to directly run CMake and update its project files.

8.3.15 Automatic loading and saving of plugin parameters

itom has the optional ability, that all plugin parameters (type *Param*), which are part of the **m_params** map of the plugin-class and do not have the *typeNoAutoSave*-flag defined, can be stored in a plugin-specific xml-file when the plugin instance is closed. This saving is not only dependent on the plugin but also on its unique identifier. **itom** has the optional ability, that all plugin parameters (of type **Param** or **ParamBase**), which are part of the **m_params** map of the plugin-class and do not have the *typeNoAutoSave*-flag defined, can be stored in a plugin-specific xml-file when the plugin instance is closed. This saving is not only dependent on the plugin but also on its unique identifier.

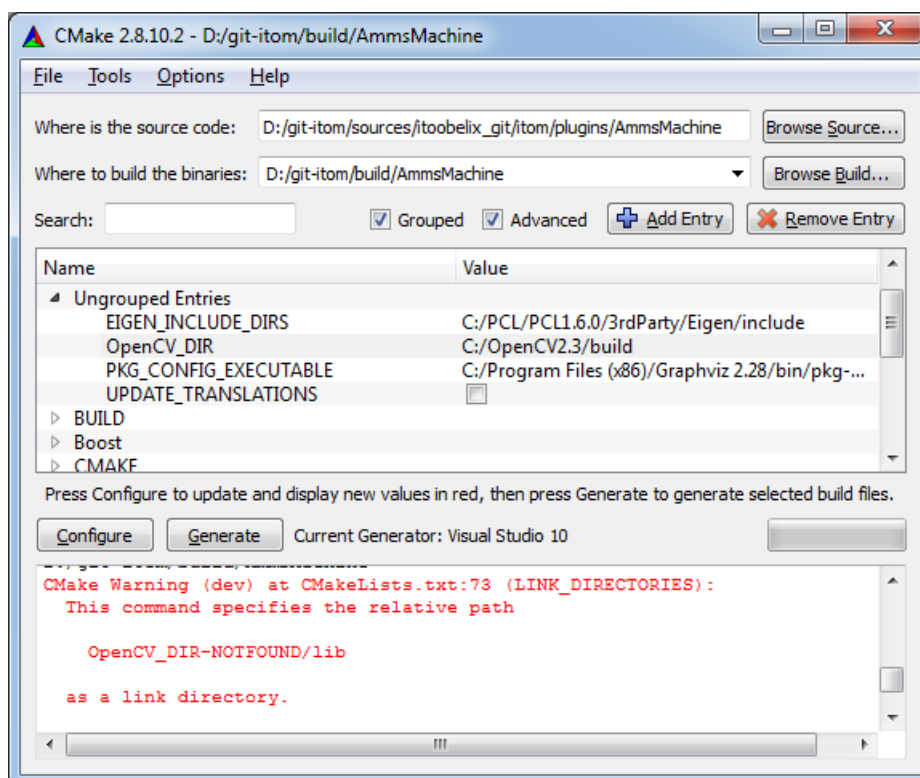
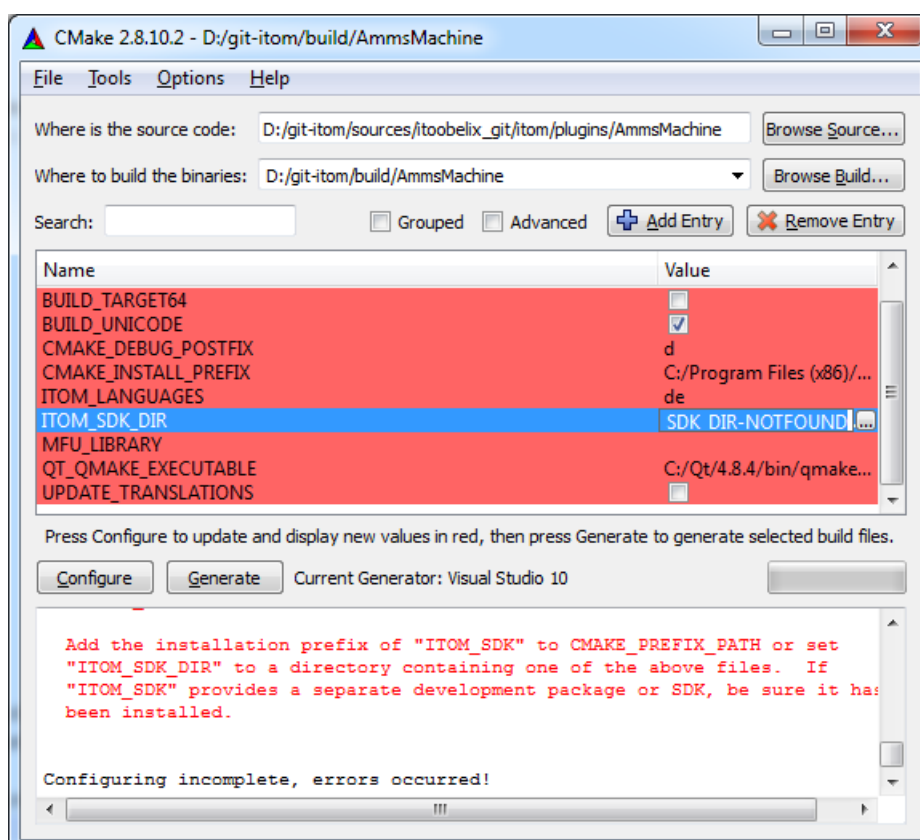
Additionally, the decision whether these parameters should be saved or not, is set by the member variable **m_autoSavePolicy** of the interface class of the plugin (see *Plugin interface class*). This variable can have the following values:

```
enum tAutoSavePolicy {
    autoSaveAlways      = 0x1, /*!< always saves parameters to xml-file at shutdown */
    autoSaveNever       = 0x2  /*!< never saves parameters to xml-file at shutdown (default)
};
```

If the member has the value *autoSaveAlways*, all parameters contained in the map *m_params* of the plugin instance are saved in a xml-subtree. This subtree is dependent on the unique identifier of the plugin. Remember, that only plugins which do not have the flag *typeNoAutoSave* will be saved (see *Parameter-Container class of itom*).

Inversely, saved parameters in a plugin specific xml-file can also be set after that the plugin instance has been created and initialized (with the mandatory and optional parameters given by the constructor in python). The loading of these xml-parameters is dependent on the value of the member variable **m_autoLoadPolicy** of the plugin's interface class. This variable can have the following values:

```
enum tAutoLoadPolicy {
    autoLoadAlways      = 0x1, /*!< always loads xml file by addInManager */
    autoLoadNever       = 0x2, /*!< never automatically loads parameters from xml-file (default)
```



```
autoLoadKeywordDefined = 0x4 /*!< only loads parameters if keyword autoLoadParams=1 exists
};
```

If the variable is set to *autoLoadAlways*, all parameters in the xml-subtree, having the same unique identifier (ID) than the recently opened plugin instance, are set in the instance. This is done by calling the method **setParam** of the instance for every specific parameter. It is up to you to decide whether you accept the given parameter from the xml-file or not. Remember that **setParam** is called after that the plugin instance is created and the init-method of the plugin has been called. Of course, the init-method gets the mandatory and optional parameters, which are indicated when you create the plugin instance for example in python.

If the variable is set to *autoLoadNever*, no parameters of the xml-file will be set to the plugin's instance.

If the variable is set to *autoLoadKeywordDefined*, the parameters from the xml-subtree are only set to the plugin's instance, if the plugin has been created in python with the last keyword-defined parameter **autoLoadParams=1**.

Example:

```
a = dataIO("DummyGrabber",param1,...,paramN,autoLoadParams=1)
```

8.3.16 itom API

Usually, plugins only have access to the sources which are defined in the **SDK** of **itom**. These are for instance the header and source files contained in the folder **common** (e.g. **addInInterface.h**) and in the folder **plot** (only relevant for developing designer plugins (hence plots, figures...)). Further, the **SDK** contains the header-files and corresponding libraries for the **dataObject** and the **pointCloud** library.

However, it is not desired at all, that plugins also include header or source files from **itom** itself. The intention is, that it should be possible to develop plugins without the need to compile **itom** from its sources or have the source code available.

Therefore, **itom** provides an **application programming interface** (API). The API has methods defined, which give plugins the possibility to use important functionality of **itom**. All available methods of the **API** are defined in the file **apiFunctionsInc.h**, that also lies in the folder **common** of the SDK. For methods concerning plots and figures, there is an additional API definition file **apiFunctionsGraphInc.h**.

Initialization

If your plugin or designer plugin widget is derived from classes **AddInBase** or **AbstractFigure**, which is usually the case, you need to do the following steps in order to use the **API** methods:

Write

```
#define ITOM_IMPORT_API
#define ITOM_IMPORT_PLOTAPI
```

at the top of the main *cpp*-file of your plugin. This definition must be before the include of the corresponding header-file and any other includes.

The **API** methods then become accessible through the files **apiFunctionsInc.h** or **apiFunctionsGraphInc.h** included in the SDK of **itom**. These files already are included in the file **addInInterface.h**. If this is included in your source file, you don't need to include the other header files. If you use **API** functions for instance in a dialog or dock widget class include **apiFunctionsInc.h** or **apiFunctionsGraphInc.h** one more time.

When you are programming a plugin, derived from **AddInBase**, you can access any **API**-method at any time in your plugin (even in the constructor). When programming a designer plugin widget that is handled as plotting plugin (derived from **AbstractFigure**) the APIs only become valid after the construction of your plugin.

Note: Due to the software structure, the valid pointer is transmitted by **itom** sending the event with type **QEvent::User+123**, that is caught by your plugin after the construction.

Usage

You can call the methods, defined in **apiFunctionsInc.h** or **apiFunctionsGraphInc.h** like normal function calls. In the following example, the filter **saveRPM**, which is defined in another plugin should be called. Since, the mandatory, optional and output parameters of this filter are unknown, we will first request their default values, then change their values and finally we call the filter-method. If you have knowledge about the parameters, you just can implement the second part.

```
ito::RetVal retVal;
QVector<ito::ParamBase> mandParams, optParams, outParams; //define three, empty parameter vectors

//now get default values
retVal = apiFilterParamBase("saveRPM", &mandParams,&optParams,&outParams);
//retVal is retOk if the filter 'saveRPM' could be found. If the external plugin does not exist, ...

//change the value of some values
mandParams[0].setVal<void*>( yourDataObjectPtr ); //set data object to save as first mandatory pa...
mandParams[1].setVal<char*>( "C:\\test.rpm" );

//now call the filter
retVal += apiFilterCall("saveRPM", &mandParams, &optParams, &outParams);

if(!retVal.containsError())
{
    //filter could be successfully executed
    //you can now evaluate the output parameters, if the filter provided some.
}
```

8.3.17 Link or load external libraries

In your plugin, you have different possibilities to use external or 3rd party libraries:

Link to a static library

If you add an appropriate entry to the linker settings of your project file, it is possible to link to external, static libraries. This is for example the case when linking to the **dataObject.lib** or **dataObject.so** (Linux) or if using any components of the *PointCloudLibrary*. When compiling your plugin, the whole implementation of the library is compiled into your plugin-library and you don't need to distribute the external file.

Link to a shared library

You can also link to an external shared library by adding the corresponding entry to the linker settings, too. This is for example the case when using *OpenCV*-methods, which use implicitly do, if you link againsts the **dataObject**. Then you must add the corresponding **.lib** or **.so** files to your linker-settings and provide the corresponding **.dll** or **.a** file (in debug or release, if possible). Then, you have to take care that **itom** is able to find the external library. Please consider, that this external library file is not detected relatively to the location of your plugin but relatively to the executable of **itom** itself. Therefore, you have these possibilities to distribute the external library file:

- Add its containing folder to the path variable of your operating system.
- Add it to any path which already is contained in the path variable of your OS (e.g. **system32** on Windows).
- Directly add it to the **itom**-folder (not recommended, since leads to “crazy” folder structure)
- Directly add it to the folder **lib** contained in the **itom**-folder. The **lib***-folder is added to the path variables that are passed to **itom** at startup. This is a good possibility to provide the external library file, however it can also lead to conflicts with other plugins, that need the same external library, however in an other version. Therefore check whether you can share the same version with other plugins. In the default implementation of **itom**, there is also some default libraries of **Glut**, **FFTW**... that should be used.

- Try to indicate the shared library as delay-loaded module. Then you also adapt the path variable of your plugin to a folder of your choice, before the plugins tries to load the shared library. This is a conflict-free way how to access shared libraries. Let's make an example: Your plugin **MyPlugin** lies after compilation in the folder **plugin/myPlugin**, that is a subfolder of **itom**. Then put your external shared library file in the subfolder **lib** of **plugin/myPlugin**. Then add this path to the current path variable of the application by adding the following code for instance in the constructor or the **init**-method of your plugin implementation:

```
// Get the path to the plugin directory
QDir dllDir = QApplication::applicationDirPath();
if( !dllDir.cd("plugins/MyPlugin") )
{
    dllDir.cdUp();
    dllDir.cd("plugins/MyPlugin");
}
dllDir.cd("lib"); //move to lib folder
QString dllDir2 = QDir::cleanPath(dllDir.filePath(""));

// Add the plugin path to the path environment variable
char *oldpath = getenv("path");
char *newpath = (char*)malloc(strlen(oldpath) + dllDir2.size() + 10);
newpath[0] = 0;
strcat(newpath, "path=");
strcat(newpath, dllDir2.toLatin1().data());
strcat(newpath, ";");
strcat(newpath, oldpath);
_putenv(newpath);
free(newpath);
```

Note: The path variable of your operating system is always copied and then passed to a newly started application. Therefore you can adapt this copy without influencing the overall path environment variable.

Load external library at runtime

The most complicated way to access an external library with respect to the programming cost is to use the command **LoadLibrary** or the platform-independent **Qt**-class **QLibrary** in order load an external library at runtime of your plugin. Then you need to resolve the symbols in the library in order to access them afterwards in a function-call. The advantage of this method however is, that the library can be at any location since you are able to load the library with its absolute filename. See the **Qt** documentation for details about the class **QLibrary**, that is recommended to use.

Note: If you link to external libraries, please consider always the license requirements of the external library.

8.3.18 Project settings for plugins

Since you are building your plugins using **CMake** most of the following settings are automatically set. However, this document gives some hints about properties, which can maybe be helpful in case of any problems.

Prerequisites

For programming a plugin, you need at least the following things:

- A C++-compatible IDE. On Windows-machines it is recommended to program with *Visual Studio Professional 2010*, since **itom** is developed with this IDE, too. On Linux-machines you can for instance use the *QtCreator* or *Eclipse*. It is difficult to develop with *Visual Studio 2010 Express*, since you should install the *Qt AddIn for Visual Studio* in order to have a good support of **Qt** within *Visual Studio*. In the case you don't have the professional version of *Visual Studio*, better consider to use the *QtCreator* for Windows. You must have the **Qt** version installed, whose major and minor version number is equal than the version **itom** has

been built with. Nevertheless the debugging of your plugin only is possible if you also have a debug-version of **itom** available on your computer, hence, you built it by yourself from the sources. Else, you only can test your plugin by extensively streaming debugging messages to the `std::cerr` or `std::cout` stream, which finally are displayed in the command line of **itom**.

- For running *itom*, you need *Python 3.2* installed on your machine.
- If you want to support the *itom*-internal *DataObject* (matrix structures), it is highly recommended to install *OpenCV2.3* or later on your machine.

Please consider to have all libraries, which you need, installed in the same version with respect to the processor type (32bit or 64bit).

General settings

- compile your plugin as dynamic library (*dll* or *a* on linux).
- for the code generation use as runtime library the setting **Multithreaded-DLL (/Md)** or **Multithreaded-Debug-DLL (/MDd)** respectively.
- Don't use any precompiled headers on Windows.
- You can switch *openMP* on in order to support multi-processor calculations for parallelizable algorithms.
- Call your DLL "[yourName].dll" for the release-version and "[yourName]d.dll" for the debug-version.

Qt-dependent settings

itom is written in C++ using the **Qt**-framework. **Qt** provides platform-independent modules and classes which extend the possibilities of native C++. For example, **Qt** gives the opportunity to build GUI-applications, have network and graphics support or to establish a platform-independent plugin system.

On the one hand, some functionalities of **Qt** can be used by the help of native Qt-applications, like the designer to build "what-you-see-is-what-you-get" user interfaces, the translator to create translations of the application..., on the other hand C++ is enlarged by **Qt** by writing specific pre-compiler commands in the code. In both cases, these features have to be translated into native C++-code during the pre-compiling process. Therefore the project files have to be adapted, such that the **Qt**-specific pre-steps will be triggered once the project's compilation process is started. All this is done if you install the **Qt-AddOn** for Visual Studio (if developping with Visual Studio IDE).

Since the plugin, you will write, is based on *Qt*'s plugin system, these steps also have to be added to the plugin's pre-compiling steps. This can be realized by different ways:

1. You use the **QtCreator** as IDE and everythings works fine (if the path to **Qt** is contained in the path variable and the environmental variable *QTDIR*)
2. You can use the professional version of Visual Studio together with the installed add-in **Qt Visual Studio Add-In**
3. You can use any other development environment and you have to add the necessary pre-compilation step by yourself in the appropriate project file.

The pre-processor-step contains the following steps:

1. In a folder "generated-files" additional files will be created for each class, containing the macro *Q_OBJECT* (moc-process).
2. Any user-interface file (*.ui) will be transformed into an additional C++-class file, that is also contained in the "generated-files" folder (uic-process).
3. The translation tables will be created.
4. The resource-files will be parsed and an appropriate C++-file is created (rcc-process).

Qt is shipped with a number of different libraries (lying in the folder *\$QTDIR\$bin*). You must link your application against the libraries, whose function you will need in your plugin. It is always necessary to link against the library **QtCore** and **QtGui** if your plugin contains any user interface functionality. Other important libraries

are **QtOpenGL** for OpenGL-support, **QtSvg** for Svg-support or **QtXml**, **QtSql** or **QtNetwork**. For each of these libraries you plugin must have an entry in the *include*-directories and the *linker*-commands.

The pre-processor-definitions must contain the following entries: * WIN32 or _WIN64 * QT_LARGEFILE_SUPPORT * QT_PLUGIN * QT_DLL * QDESIGNER_EXPORT_WIDGETS

and for every Qt-library you need (in capital letters) * QT_CORE_LIB * QT_GUI_LIB * ...

Include settings

Add the following include-directories (\$(ITOM_QTDIR) is the path to the **Qt**-source directory - having subfolders like include or bin):

- .GeneratedFiles
- .GeneratedFilesDebug or Release
- \$(ITOM_QTDIR)include
- \$(ITOM_QTDIR)includeqtmmain

and for every further Qt-library * \$(ITOM_QTDIR)includeQtCore * \$(ITOM_QTDIR)includeQtGui * ...

Additionally you should add the path to OpenCV's include directory, if you are using or linking against the *DataObject*.

Linker settings

Add as linker directory:

- The directory, where itom is saving the library to the dataObject.lib...
- The library-path of OpenCV
- The directory \$(ITOM_QTDIR)bin

Your plugin should at least link againsts the following libraries:

DEBUG:

- qtmmaind.lib
- QtCored4.lib
- QtGui4.lib
- ...further Qt libraries
- opencv_core\$(ITOM_OPENCV_VER)d.lib
- DataObjectd.lib
- ...

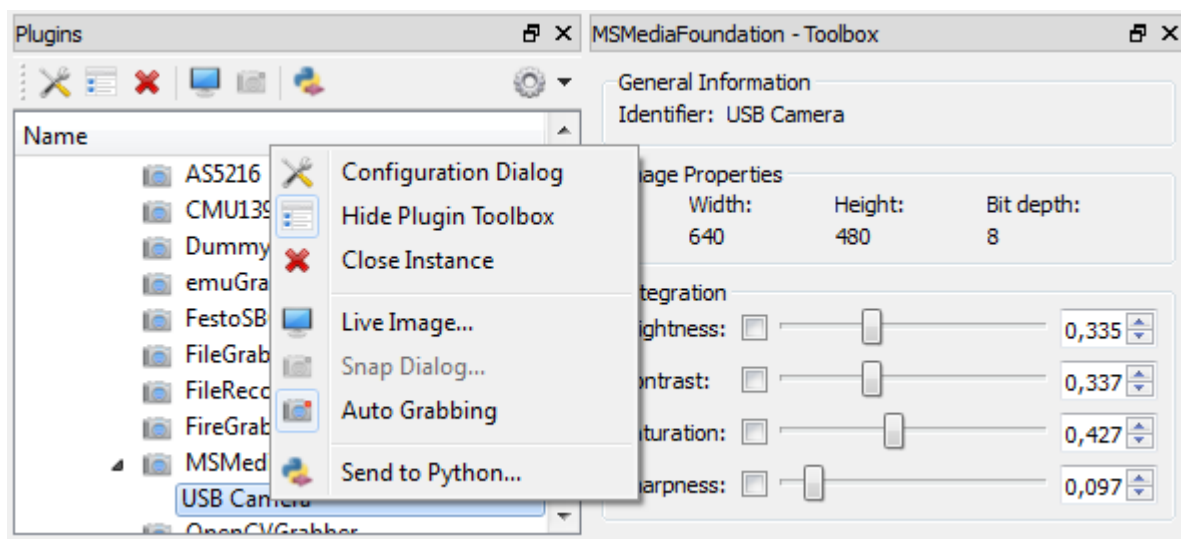
RELEASE:

- qtmmain.lib
- QtCore4.lib
- QtGui4.lib
- ...further Qt libraries
- opencv_core\$(ITOM_OPENCV_VER).lib
- DataObject.lib
- ...

Note: For more information about the deployment of plugins, including notes about the **Qt**-version compatibility, see [link to Qt-documentation](#)

8.3.19 Optional dock widget (toolbox) for hardware plugins

Every hardware plugin (actuator, dataIO) can provide one toolbox (alternatively called *dockWidget*) that can be docked into the main window of **itom** and provide fast access to commonly used parameters and settings of the plugin. An example for the toolbox of a USB camera device of the plugin *MSMediaFoundation* looks like:



Usually, the toolbox can be opened by the menu or context menu of the plugin toolbox of **itom**. Alternatively, the plugin classes in Python provide the methods:

```
cam.showToolbox()
motor.hideToolbox()
```

This section describes a possibility to generate such a toolbox.

Base idea behind the toolbox

Before you start programming the toolbox, consider the following hints:

- Although the plugin is always executed in its own thread, the toolbox always runs in the main thread of **itom**. Qt has a general restriction that GUI related things must always be executed in the main thread.
- Therefore the communication between toolbox and plugin must be implemented via a thread-safe signal/slot mechanism.
- **itom** provides the base class **ito::AbstractAddInDockWidget** in order to unify and simplify this process. This base class is contained in the SDK.
- Usually you will design the toolbox with the **Qt Designer** and implement the specific code within a class that is inherited from **ito::AbstractAddInDockWidget**.
- As last step, you need to create one instance of the toolbox class within the constructor of the plugin itself and register the toolbox such that **itom** knows about its existence.

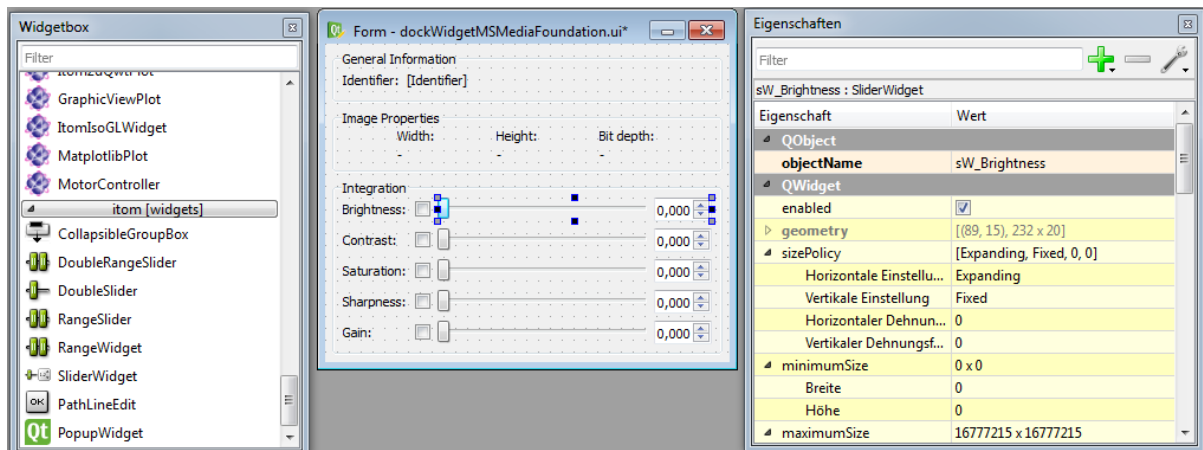
User Interface Design

The base design of the toolbox is done in the **Qt Designer**. Its output is a **ui-file**, that is integrated in your plugin project. Within this **ui-file** it is possible to use specific widgets from the **itomWidgets** project. For instance there

exist sliders for double values or sliders with two bar such that a minimum and maximum value can be adjusted using the slider. If you want to have access to these widgets (besides the common Qt widgets like buttons, checkboxes,...), open the **Qt Designer** from **itom**.

Then create a new form of type **Widget** (you are asked this when clicking the new button in Qt Designer). Next you can design the widget using items from the widgetbox. Don't forget to set appropriate layouts in order to have a nice, re-scalable widget. Furthermore, you should try to design a small widget, since it should fit inside the main window of **itom** (e.g. use tabs, ...).

The following image shows the example of the toolbox from the **MSMediaFoundation** plugin where the double sliders from the **itomWidgets** component is used.



Don't forget to give all widgets a suitable object name and configure their properties as far as you can do this at that moment. The final configuration is done later in the C++ class and can therefore be adjusted at runtime, e.g. depending on the specific device. One last thing to remember is to give the overall widget a suitable object name as well (like **DockWidgetMSMediaFoundation** in the example). Then save the **ui**-file in the source folder of your plugin. In the example the filename is **dockWidgetMSMediaFoundation.ui**.

The **ui**-file can only be used and interpreted in your plugin source code if it is inserted into the **CMakeLists.txt** file of the plugin in the right way. The most important things, that are necessary therefore are contained in the following CMake snippet. For more information see the documented template **CMakeLists.txt** files in the *pluginTemplates* folder of the itom source code.

```
#... somewhere in the area where the header and source files are inserted

set(plugin_UI
    #add absolute pathes to any *.ui files
    ${CMAKE_CURRENT_SOURCE_DIR}/dockWidgetMSMediaFoundation.ui
)

#parses the ui-file and generates the corresponding C++ moc-file
if (QT5_FOUND)
    QT5_WRAP_UI(plugin_UI_MOC ${plugin_UI})
else (QT5_FOUND)
    QT4_WRAP_UI_ITOM(plugin_UI_MOC ${plugin_UI})
endif (QT5_FOUND)

#...

ADD_LIBRARY(${target_name} ... ${plugin_UI_MOC} ...)
```

Necessary Source Code

After having created the basic user interface in Qt Creator, the toolbox now consists of another header and source file. Create the two files in the source directory of the plugin and insert them in the list of header and source files in the plugin's **CMakeLists.txt**. The header file should look like this:

```
#ifndef DOCKWIDGETYOURPLUGIN_H
#define DOCKWIDGETYOURPLUGIN_H

#include "common/abstractAddInDockWidget.h"
#include "common/addInInterface.h"

#include <qmap.h>
#include <qstring.h>

#include "ui_filenameOfTheToolboxUiFile.h" //TODO

class DockWidgetYourPlugin : public ito::AbstractAddInDockWidget
{
    Q_OBJECT

public:
    DockWidgetYourPlugin(ito::AddInBase *pluginInstance);
    ~DockWidgetYourPlugin() {};

private:
    Ui::ObjectNameOfTheUserInterface ui; //TODO
    bool m_inEditing;
    bool m_firstRun;

public slots:
    void parametersChanged(QMap<QString, ito::Param> params);
    void identifierChanged(const QString &identifier);

    ///  
//< for actuators add the following two lines  
//void actuatorStatusChanged(QVector<int> status, QVector<double> actPosition);  
//void targetChanged(QVector<double> targetPositions);

private slots:
    /*define slots that are called if values of the widgets  
are changed. The connection of the valueChanged, clicked...  
signals to these slots can automatically be done using the  
Qt's auto-connection syntax.

    Example for a slot invoked if the value of the slider sW_Brightness  
changed:
    */  
// void on_sW_Brightness_valueChanged(double d);
};

#endif
```

Some words about this header file:

- The constructors obtains the pointer to the plugin itself as argument.
- The member **ui** is a reference to the auto-created class of the **ui**-file. By this member you get access to all widgets added in Qt Creator.
- The member **m_inEditing** is used to avoid a never ending ring of “widget value changed” -> “change parameter in plugin” -> “inform toolbox about change” -> “change widget”...
- The member **m_firstRun** can be used to check if the parameters (m_params) of the plugin are send to the toolbox for the first time in order to initialize/configure some widgets at the first run.

- The slot **parametersChanged** should be connected to the plugin's signal with the same name in the **dock-WidgetVisibilityChanged** method of the plugin. It is called whenever a parameter of the plugin has been changed.
- The slot **identifierChanged** is invoked if the identifier of the plugin has been changed using **ito::AddInBase::setIdentifier(...)** (e.g. in the init method of the plugin).
- The slots **actuatorStatusChanged** and **targetChanged** (actuators only) should be connected to the signals with the same name of the plugin identical to **parametersChanged**. They are invoked if the status, current position or target position of an actuator axis changed.
- adapt the lines marked with **//TODO**.

Now, some hints about the implementation of the different methods in the source file.

The constructor passes the pluginInstance pointer to the constructor of the super class **AbstractAddInDockWidget** and initializes the ui-file:

```
DockWidgetYourPlugin::DockWidgetYourPlugin(ito::AddInBase *pluginInstance) :
    AbstractAddInDockWidget(pluginInstance),
    m_inEditing(false),
    m_firstRun(true)
{
    ui.setupUi(this); //initialize ui-file and auto-connect slots

    //for slider widgets it is convenient to set their
    //tracking property to false, such that valueChanged
    //is only emitted if the input ends and not after every
    //single character (e.g. 3 emits for the input of '100')
    //Example:
    //ui.sW_Brightness->setTracking(false);
}
```

Initialize the widgets depending on the parameters of the plugin and change their current value if the parameters of the plugin changed (e.g. by a python script):

```
void DockWidgetYourPlugin::parametersChanged(QMap<QString, ito::Param> params)
{
    if (m_firstRun)
    {
        //use params (identical to m_params of the plugin)
        //and initialize all widgets (e.g. min, max values, labels, enable some,...)
        m_firstRun = false;
    }

    if (!m_inEditing)
    {
        m_inEditing = true;

        //change the current value of all widgets to the value given in the params map

        m_inEditing = false;
    }
}
```

In the **identifierChanged** slot, the current string identifier of the plugin instance is passed (usually one time). If you have a label for this, adjust the text property of this label widget to the given string. Example:

```
void DockWidgetYourPlugin::identifierChanged(const QString &identifier)
{
    ui.lblIdentifier->setText(identifier);
}
```

Finally, you only need to implement the slots invoked if the user changes any values of the widgets. In a toolbox, there is no Apply or OK button like in a configuration dialog. Therefore, the plugin should immediately react on

changes of the widgets. An example for the brightness slider is:

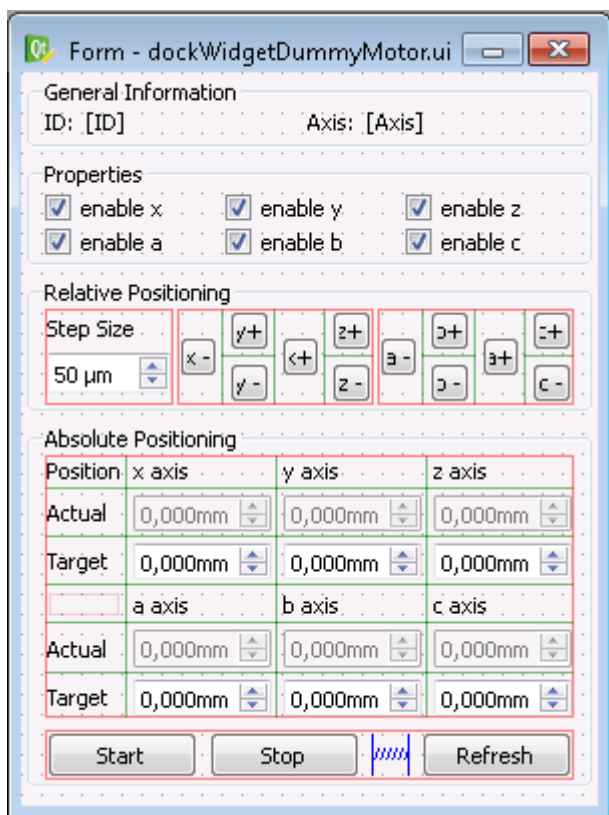
```
void DockWidgetYourPlugin::on_sW_Brightness_valueChanged(double d)
{
    //d is the new value

    if (!m_inEditing) //only send the value to the plugin if not inEditing mode
    {
        m_inEditing = true;
        QSharedPointer<ito::ParamBase> p(new ito::ParamBase("brightness", ito::ParamBase::Double, d));
        setPluginParameter(p, msgLevelWarningAndError);
        m_inEditing = false;
    }
}
```

In this function, a `QSharedPointer` of `ito::ParamBase` is created with a new instance of the brightness parameter, that contains the value `d`. This parameter `p` is then passed to the plugin instance (via signal/slot connection). All this is done in the method `setPluginParameter`, defined in the base class `AbstractAddInDockWidget`. See the documentation there for more information and further methods, e.g. for passing multiple parameters. If the second argument is set to `msgLevelWarningAndError`, all warnings and errors from the `setParam` method of the plugin instance are observed and a message box appears. `setPluginParameter` finishes if the result of `setParam` of the plugin is received.

Special functions for actuators

Usually, the toolboxes of actuators should contain methods to relatively move single axes by a certain distance or to set the target positions of all axes and start the movement by a start button. Once the start button is clicked, a stop button can appear whose click signal forces the interrupt of the plugin. The following image shows the ui-file of the plugin **DummyMotor**:



In difference to the implementation given above, you need to learn

- how to start an axis movement

- how to abort the movement
- how to change the appearance of the widgets depending on the state of the axes (e.g. yellow background if moving, red background in case of error and default, white background else)

Lets start with interrupting the movement of all active axes. Simply call the method

```
setActuatorInterrupt();
```

defined in **AbstractAddInDockWidget**.

For relatively or absolutly moving one or multiple axes, directly call the method

```
setActuatorPosition(QVector<int> axes, QVector<double> positions, \
    bool relNotAbs, MessageLevel msgLevel = msgLevelWarningAndError)
//or
setActuatorPosition(int axis, double position, bool relNotAbs, \
    MessageLevel msgLevel = msgLevelWarningAndError)
```

also defined in **AbstractAddInDockWidget**. They invoke the slots **setPosAbs** or **setPosRel** of the actuator plugin, wait for the release of the semaphore. **relNotAbs** decides whether the movement is absolut or relative. Use the **msgLevel** to let a message box appear if the movement failed or returned a warning. The default value does both.

Finally, you should implement and connect the slot **targetChanged** and **actuatorStatusChanged** (see header file above). In **targetChanged**, set the value of the related widgets of the user interface to the new target values (in *mm* or *degree*). The slot **actuatorStatusChanged** is used to update the current positions as well as to analyze the status of all axes.

A dummy implementation changes the background color of the current-value spin boxes for all axes depending on their state. It looks somehow like this:

```
void DockWidgetYourPlugin::actuatorStatusChanged(QVector<int> status, QVector<double> positions)
{
    bool running = false;
    QString style;

    for (int i = 0; i < status.size(); i++)
    {
        if (status[i] & ito::actuatorMoving)
        {
            style = "background-color: yellow";
            running = true;
        }
        else if (status[i] & ito::actuatorInterrupted)
        {
            style = "background-color: red";
        }
        /*else if (status[i] & ito::actuatorTimeout)
        {
            style = "background-color: green";
        }*/
        else
        {
            style = "background-color: ";
        }

        //here: m_spinCurrentPos is a member pointing to different spin boxes
        // displaying the current position of a specific axis. Do it like it is
        // convenient for you
        m_spinCurrentPos[i]->setStyleSheet(style);
    }

    //in the demo plugin, a method 'enableWidget' with a boolean
    //argument is implemented to en/disable all moving related widgets
    //depending if the motor is running or not (and hides/shows the
```

```
//start and stop buttons.
enableWidget(!running);

for (int i = 0; i < positions.size(); i++)
{
    m_spinCurrentPos[i]->setValue(positions[i]);
}
}
```

Prepare the plugin for the toolbox

One last thing needs to be done in order to register the toolbox in the plugin class itself.

At first, insert the following lines at the end of the constructor of the plugin class in order to create one instance of the toolbox class and register the toolbox within a dockable toolbox of the main window of itom:

```
DockWidgetYourPlugin *toolbox = new DockWidgetYourPlugin(this);

Qt::DockWidgetAreas areas = Qt::AllDockWidgetAreas; //areas where the toolbox can be positioned (

//define some features, saying if the toolbox can be closed, can be undocked (floatable) and move
QDockWidget::DockWidgetFeatures features = QDockWidget::DockWidgetClosable | \
    QDockWidget::DockWidgetFloatable | QDockWidget::DockWidgetMovable;

//register the toolbox
createDockWidget(QString(m_params["name"].getVal<char *>()), features, areas, toolbox);
```

The last thing: Overload and implement the slot **dockWidgetVisibilityChanged** of the plugin, originally defined in **ito::AddInBase**. This is called whenever the plugin gets visible or is hidden. If it is visible, connect the signal/slot **parametersChanged** as well as **targetChanged** and **actuatorStatusChanged** in case of an actuator. This reduced the things to do if the toolbox is not connected. If it gets connected again, forces the resubmission of the current set of parameters, target positions... See the following example how to do this:

```
void YourPlugin::dockWidgetVisibilityChanged(bool visible)
{
    if (getDockWidget())
    {
        QWidget *w = getDockWidget()->widget(); //your toolbox instance
        if (visible)
        {
            QObject::connect(this, SIGNAL(parametersChanged(QMap<QString, ito::Param>)), w, \
                SLOT(parametersChanged(QMap<QString, ito::Param>)));
            emit parametersChanged(m_params); //send current parameters

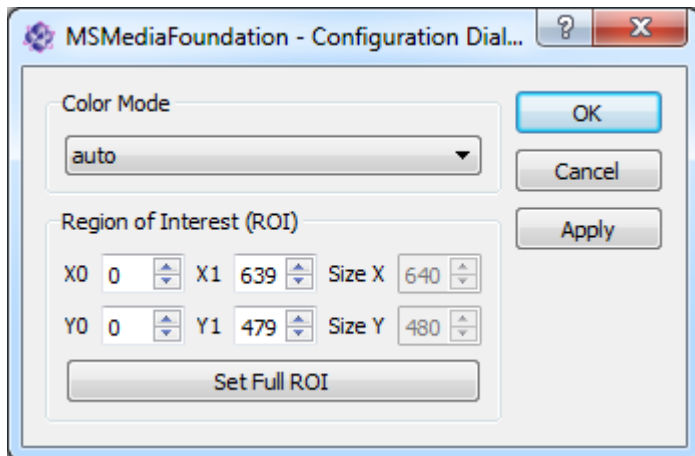
            //actuators only
            QObject::connect(this, SIGNAL(actuatorStatusChanged(QVector<int>,QVector<double>)), w, \
                SLOT(actuatorStatusChanged(QVector<int>,QVector<double>)));
            QObject::connect(this, SIGNAL(targetChanged(QVector<double>)), w, \
                SLOT(targetChanged(QVector<double>)));
            requestStatusAndPosition(true,true); //send current status, positions and targets
        }
        else
        {
            QObject::disconnect(this, SIGNAL(parametersChanged(QMap<QString, ito::Param>)), w, \
                SLOT(parametersChanged(QMap<QString, ito::Param>)));

            //actuators only
            QObject::disconnect(this, SIGNAL(actuatorStatusChanged(QVector<int>,QVector<double>)), w, \
                SLOT(actuatorStatusChanged(QVector<int>,QVector<double>)));
            QObject::disconnect(this, SIGNAL(targetChanged(QVector<double>)), w, \
                SLOT(targetChanged(QVector<double>)));
        }
    }
}
```

```
}
}
```

8.3.20 Optional configuration dialog for hardware plugins

Every hardware plugin (actuator, dataIO) can provide one configuration dialog that can be shown as a modal dialog (if shown, the remaining application is blocked until the dialog is closed). Using this dialog the user can configure the plugin's instance without need of scripting. An example for the configuration dialog of a USB camera device of the plugin *MSMediaFoundation* looks like:



In difference to the optional *toolboxes*, the parameter is not directly changed once the user changes the value of any widget, but the configuration dialog should provide an OK, cancel and optional apply button. By clicking OK, all currently changed values are applied and sent to the plugin via multiple calls to **setParam**. Apply has the same behaviour but the dialog is not closed. By clicking cancel or closing the dialog, no changes are applied.

Usually, the configuration dialog can be opened by the menu or context menu of the plugin toolbox of **itom**. Alternatively, the plugin classes in Python provide the methods:

```
cam.showConfiguration()
```

This section describes a possibility to generate such a toolbox.

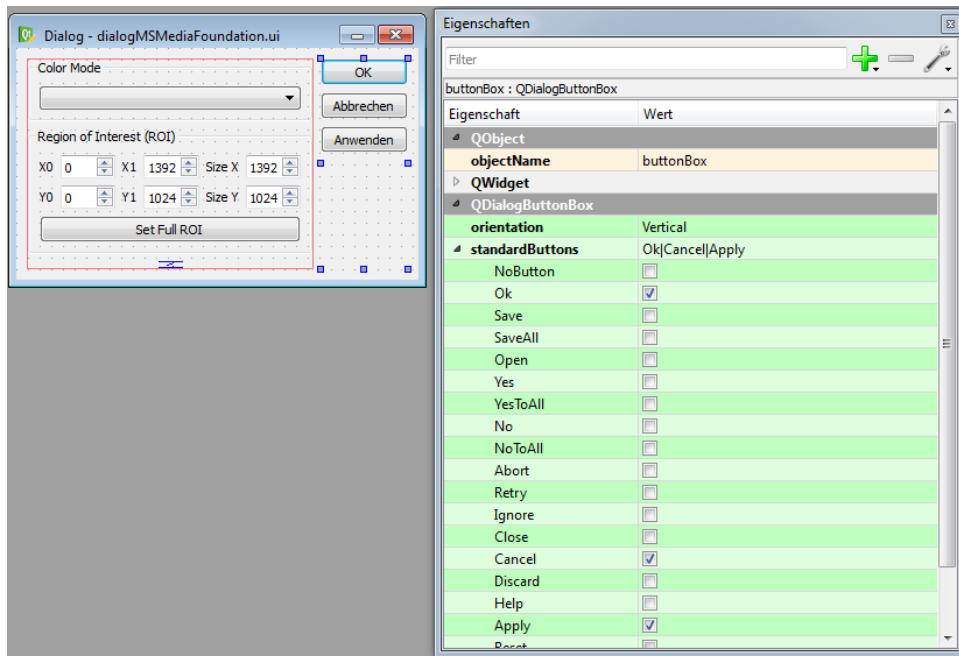
Base idea behind the config dialog

Before you start programming the dialog, read the documentation about how to program a *toolbox* and consider the following specific hints:

- Although the plugin is always executed in its own thread, the configuration dialog always runs in the main thread of **itom**. Qt has a general restriction that GUI related things must always be executed in the main thread.
- Therefore the communication between config dialog and plugin must be implemented via a thread-safe signal/slot mechanism.
- **itom** provides the base class **ito::AbstractAddInConfigDialog** in order to unify and simplify this process. This base class is contained in the SDK.
- Usually you will design the config dialog with the **Qt Designer** and implement the specific code within a class that is inherited from **ito::AbstractAddInConfigDialog**.
- As last step, you need to add some lines of code in the plugin class such that the plugin and itom know about the existence of the config dialog.
- Since the configuration dialog usually does not provide functionalities to move any actuator axes, the programming for dataIO and actuator devices is the same (in difference to the *toolboxes*).

User Interface Design

The base design of the config dialog is done in **Qt Designer**. The user is referred to the documentation about creating *toolboxes* in order to learn how to create the user interface. The dialog is not created as widget but as dialog. As you see in the following screenshot



it is convenient to realize the OK, Cancel and Apply buttons as widget **QDialogButtonBox**. Click the Ok, Cancel and Apply items of the **standardButtons** property in order to show the desired buttons. It is also allowed to use single buttons for the same behaviour. However, in the following, the dialog button box is used.

Necessary Source Code

After having created the basic user interface in Qt Creator, the config consists of a header and source file. Create the two files in the source directory of the plugin and insert them in the list of header and source files in the plugin's **CMakeLists.txt**. The header file should look like this:

```
#ifndef DIALOGYOURPLUGIN_H
#define DIALOGYOURPLUGIN_H

#include "common/param.h"
#include "common/retVal.h"
#include "common/sharedStructuresQt.h"
#include "common/abstractAddInConfigDialog.h"

#include "ui_nameOfTheDialogUiFile.h" //TODO

#include <qstring.h>
#include <qmap.h>
#include <qabstractbutton.h>

namespace ito
{
    class AddInBase; //forward declaration
}

class DialogYourPlugin : public ito::AbstractAddInConfigDialog
{

```

```

Q_OBJECT

public:
    DialogYourPlugin(ito::AddInBase *pluginInstance);
    ~DialogYourPlugin() {};

    ito::RetVal applyParameters();

private:
    bool m_firstRun;
    Ui::ObjectNameOfTheUIDialog ui; //TODO

public slots:
    void parametersChanged(QMap<QString, ito::Param> params);

private slots:
    //auto-connected slot called if ok, apply or cancel is clicked
    void on_buttonBox_clicked(QAbstractButton* btn);

    /*Since all parameters are only evaluated and sent
    if one of the buttons are clicked, you usually don't
    need to connect to the specific signals of all widgets.
    However, if you need to implement special interactive
    functionalities, define some further slots here.
    */
};

#endif

```

The header file has an easier structure than the class of a toolbox, since parameters are only evaluated and sent if one of the buttons are clicked. Some words about this header file:

- The constructors obtains the pointer to the plugin itself as argument.
- The member **ui** is a reference to the auto-created class of the **ui**-file. By this member you get access to all widgets added in Qt Creator.
- The member **m_firstRun** can be used to check if the parameters (**m_params**) of the plugin are send to the config dialog for the first time in order to initialize/configure some widgets at the first run.
- The slot **parametersChanged** is called if some parameters of the plugin are (externally) changed and is called after the creation of the config dialog (see *firstRun*).
- The method **applyParameters()** is responsible for sending all changed parameters to the plugin. This method is automatically called if the dialog is closed using the **apply**-signal (done by the plugin). If the apply button has been clicked, the programmer should call this method as well.
- adapt the lines marked with **//TODO**.

Now, some hints about the implementation of the different methods in the source file.

The constructor passes the **pluginInstance** pointer to the constructor of the super class **AbstractAddInDockWidget** and initializes the **ui**-file:

```

DialogYourPlugin::DialogYourPlugin(ito::AddInBase *pluginInstance) :
    AbstractAddInConfigDialog(pluginInstance),
    m_firstRun(true)
{
    ui.setupUi(this);
}

```

Initialize the widgets depending on the parameters of the plugin and change their current value if the parameters of the plugin changed (e.g. by a python script):

```
void DialogYourPlugin::parametersChanged(QMap<QString, ito::Param> params)
{
    //use params (identical to m_params of the plugin)
    //and initialize all widgets (e.g. min, max values, labels, enable some,...)

    //if you use two range widgets (class RangeWidget from itomWidgets) for visualizing the ROI,
    //you can directly pass the constraints of the width and height in terms of a ito::RectMeta st.
    //to the plugin parameter 'roi' to the RangeWidgets:
    /*ito::RectMeta *rm = static_cast<ito::RectMeta*>(params["roi"].getMeta());
    ui.rangeX01->setLimitsFromIntervalMeta(rm->getWidthRangeMeta());
    ui.rangeY01->setLimitsFromIntervalMeta(rm->getHeightRangeMeta());*/

    //change the current value of all widgets to the value given in the params map

    m_currentParameters = params;
}
```

Note: In difference to the **parametersChanged** method of the toolboxes, the current set of parameters is saved in the member **m_currentParameters**, defined in **ito::AbstractAddInConfigDialog**. This can then be used to check if a parameter has been changed or not. Only changed parameters should be sent back to the plugin in the **applyParameters** method.

Here is the code for a click on any button of the button box (the **objectName** of the button box in Qt Designer is **buttonBox**:

```
void DialogYourPlugin::on_buttonBox_clicked(QAbstractButton* btn)
{
    ito::RetVal retValue(ito::retOk);

    QDialogButtonBox::ButtonRole role = ui.buttonBox->buttonRole(btn);

    //cancel button, emit reject() -> dialog is closed
    if (role == QDialogButtonBox::RejectRole)
    {
        reject(); //close dialog with reject
    }
    //ok button, emit accept() -> dialog is closed
    else if (role == QDialogButtonBox::AcceptRole)
    {
        accept(); //AcceptRole
    }
    else //apply button, only call applyParameters
    {
        applyParameters(); //ApplyRole
    }
}
```

The most important function of these configuration dialogs is the method **applyParameters**. In the following example, these things are mainly done:

- A vector of **QSharedPointer<ito::ParamBase>**, called *values* is created
- Every parameter that has been changed is added to this vector with its right name, type and new value
- The vector is sent to the plugins instance using the method **setPluginParameters**, defined in **ito::AbstractAddInConfigDialog**.
- The method usually shows a message box if an error or warning occurs during the *setParam* process (see value *msgLevelWarningAndError*).

```
ito::RetVal DialogYourPlugin::applyParameters()
{
    ito::RetVal retValue(ito::retOk);
```

```

    QVector<QSharedPointer<ito::ParamBase> > values;
    bool success = false;

    //only send parameters which are changed

    //example for 'colorMode', shown as comboBox in the configDialog
    if (QString::compare(m_currentParameters["colorMode"].getVal<char*>(), ui.comboBoxColorMode->currentText().toLatin1().data()) != 0)
    {
        values.append(QSharedPointer<ito::ParamBase>(new ito::ParamBase("colorMode", ito::ParamBase::ColorMode, ui.comboBoxColorMode->currentText().toLatin1().data())));
    }

    //check further parameters...

    retVal += setPluginParameters(values, msgLevelWarningAndError);

    return retVal;
}

```

Implement the config dialog in the plugin

In order to add the configuration dialog in the plugin class, two things need to be done.

1. Overload the method **hasConfDialog** in the public domain of the plugin class (e.g. header file) and return 1 in order to indicate that this plugin has a configuration dialog and define the method **showConfDialog**:

```

class YourPlugin : public ...
{
    ...
    public:
        const ito::RetVal showConfDialog(void);
        int hasConfDialog(void) { return 1; }
    ...
}

```

2. Implement **showConfDialog** in the plugin's source file:

```

const ito::RetVal YourPlugin::showConfDialog(void)
{
    return apiShowConfigurationDialog(this, new DockWidgetYourPlugin(this));
}

```

The api method in the **showConfDialog** method mainly connects the **parametersChanged** signal of the plugin with the same slot in the configuration dialog and sends the current set of parameters (*m_params*) of the plugin to the dialog. Next, the dialog is executed as modal dialog. The function waits until either the **apply** or **reject** signal of the dialog is sent and the dialog is closed. If the **apply** signal is sent, **applyParameters** of the dialog is called such that the parameters of the dialog are set in the plugin. Last but not least, the dialog is destroyed.

PYTHON SCRIPTING LANGUAGE

9.1 Introduction

One main component of **itom** is the integrated scripting language **Python**. In order to make **itom** ready for the future, **Python** is based on the version-family 3. If you already know **Python** or already have used **Python**, you probably know about the huge functionality which already is provided by **Python** itself or by one of its various modules, that are available in the internet. The good is, that you can execute the same **Python**-scripts in **itom**. There are only few limitations concerning some **Python** modules (see *Python Limitations in itom*).

If you are a **Python** beginner, you probably should have a look at our tutorial page that also contains links to recommended tutorials in the internet:

↳.

9.1.1 Python tutorial

The goal of this tutorial is to give prospective itom users a quick overview about the basic **Python v3** commands and how to use them within **itom**.

Content

↳.

Creating a project folder and run a first program

Create a project folder

Keeping track of more difficult programs is inconvenient when only using the **litoml** *terminal (command line)*. That's why you will learn how to use script files in this chapter. Basically, script files are just regular text files that happen to have the extension *.py*.

We will store all our script files in a project folder within the **litoml** installation directory called *python_tutorial*. This folder can be created within **litoml** by right-clicking on the **litoml** folder in the *file system toolbox* and choosing *create new folder*. If you've used the default installation directory, you should end up with a folder like this: *C:\Program Files\itom\python_tutorial*.

Creating a .py file and running it

Now create a new python file in *C:\Program Files\itom\python_tutorial* and rename the default file name (*New Script.py*) to *Hello World.py*. When double-clicking the file name, the *script editor* will pop up, which we will use for all upcoming examples. For a start, type the following line of code into the script editor and save the file.

```
1 print("Hello World")
```

To run the code you've just inserted, press *F5* or click on the *run* button.

As a result, you will see the output 'Hello World' in the **litom** terminal. As you've probably figured out, the command `print()` is used to output certain characters and, hence, can be used to give the user some sort of feedback. Please note that the print command changed from earlier 2.X versions of **Python**.

Encoding of files

If your script file contains non ASCII characters (like German special characters), you will probably get an encoding error when trying to run or debug this script. This is due to the fact, that **Python** interprets your script as default utf8 text file. If you used another encoding for your file you need to tell this to **Python**, e.g. by putting

```
# -- coding: iso-8859-15 --
```

at the first line of your script file (iso-8859-15 represents the Western Europe charset). For more information see <https://www.python.org/dev/peps/pep-0263/>.

İ>ç.

Variables and names

Using variables

Now you can output things with `print()` and you can do math. The next step is to learn about variables. In programming, a variable is nothing more than a name for something and helps making the code easier to read. In **Python** the equal sign (=) is used to assign a value to a variable.

```
1 width = 10
2 depth = 5
3 height = 20
4
5 area_size = width * depth
6 volume = area_size * height
7
8 # Variables can be output with print()
9 print(area_size)
10
11 # The output of strings and variables can be combined
12 print("Volume:", volume)
```

```
50
Volume: 1000
```

A value can be assigned to several variables simultaneously as seen below. Additionally, you will learn how to make strings that have variables embedded in them. You embed variables inside a string by using specialized format sequences (in this case %d) and then putting the variables at the end with a special syntax.

```
1 width = depth = height = 10
2
3 print("The edge lengths of the cube are %d, %d and %d, respectively." % (width, depth, height))
4 print("Still the same volume: %d" % (width * depth * height))
```

```
The edge lengths of the cube are 10, 10 and 10, respectively.
Still the same volume: 1000
```

Note: Remember to put `# -- coding: utf-8 --` at the top of your script file if you use non-ASCII characters and get an encoding error.

İ>ç.

Strings and text

Strings

Besides numbers, **Python** can also manipulate strings, which can be expressed in several ways. A string is usually a bit of text you want to display to someone, or “export” out of the program you are writing. They can be enclosed in single quotes or double quotes:

```
1 print('It works.')
2 print('It doesn\'t matter.')
3 print("It doesn't matter.")
4 print('"Yes", he said.')
5 print("\\"Yes,\" he said.")
6 print('It\'s not", she said.')
```

```
It works.
It doesn't matter.
It doesn't matter.
"Yes", he said.
"Yes," he said.
"It's not", she said.
```

String formatting

Additionally, Strings may contain format characters such as `%d` from the previous section to output or convert to integer decimals. Here are some more:

Conversion	Meaning
<code>%d</code> or <code>%i</code>	Signed integer decimal
<code>%x</code> or <code>%X</code>	Signed hexadecimal (lowercase/uppercase)
<code>%e</code> or <code>%E</code>	Floating point exponential (lowercase/uppercase)
<code>%f</code> or <code>%F</code>	Floating point decimal format
<code>%c</code>	Single character (accepts integer or single character string)
<code>%r</code>	String (converts any Python object using <code>repr()</code>)
<code>%s</code>	String (converts any Python object using <code>str()</code>)

Here are some examples to try for yourself:

```
1 x = "There are %d types of people." % 10
2 binary = "binary"
3 do_not = "don't"
4 y = "Those who know %s and those who %s." % (binary, do_not)
5
6 print(x)
7 print(y)
8
9 print("I said: %r." % x)
10 print("I also said: '%s'." % y)
```

```
There are 10 types of people.
Those who know binary and those who don't.
```

Note: The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer `str.format()` interface helps avoiding these errors. Check the official [documentation](#) for more on the topic.

More on strings

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```
1 hello = "This is a rather long string containing\n\
2 several lines of text just as you would do in C.\n\
3     Note that whitespace at the beginning of the line is \
4 significant."
5
6 print(hello)
```

```
This is a rather long string containing
several lines of text just as you would do in C.
Note that whitespace at the beginning of the line is significant.
```

Or, strings can be surrounded in a pair of matching triple-quotes: `"""` or `'''`. End of lines do not need to be escaped when using triple-quotes, but they will be included in the string. So the following uses one escape to avoid an unwanted initial blank line.

```
1 print("""\
2 Usage: thingy [OPTIONS]
3     -h                Display this usage message
4     -H hostname       Hostname to connect to
5 """)
```

```
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

If we make the string literal a “raw” string, `n` sequences are not converted to newlines, but the backslash at the end of the line, and the newline character in the source, are both included in the string as data. Thus, the example:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print(hello)
```

```
"This is a rather long string containing\n\several lines of text much as you would do in C."
```

Byte array....

:)
İ»¿.

Dictionaries, Lists and Tuples

Dictionaries

One of Python’s built-in datatypes is the dictionary, which defines one-to-one relationships between keys and values.

Defining Dictionaries It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{ }`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

```

1 d = {"lens": "zoom", "software": "itom"}
2 print(d)
3
4 # 'lens' is a key and its associated value is 'zoom'. It can be referenced by d["lens"].
5 print(d["lens"])
6
7 # 'software' is a key and its associated value is 'itom'. It can be referenced by d["software"].
8 print(d["software"])
9
10 # You can get values by key, but you can't get keys by value.
11 # So d["lens"] is 'zoom', but d["zoom"] raises an exception, because 'zoom' is not a key.
12 print(d["zoom"])

```

```

{'lens': 'zoom', 'software': 'itom'}
zoom
itom
Traceback (most recent call last):
  File "..", line ?, in <module>
KeyError: 'zoom'

```

Modifying Dictionaries You can not have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value. You can add new key-value pairs at any time. This syntax is identical to modifying existing values. Dictionaries have no concept of order among elements.

```

1 d = {"lens": "zoom", "software": "itom"}
2 print(d)
3
4 d["software"] = "itom"
5 print(d)
6
7 d["institute"] = "ITO"
8 print(d)
9
10 # Dictionary keys can be modified. The old value is simply replaced with a new one.
11 d["institute"] = "none"
12 print(d)
13
14 # Dictionary keys are case-sensitive. Here, a new key-value pair will be created.
15 d["Institute"] = "another"
16 print(d)

```

```

{'software': 'itom', 'lens': 'zoom'}
{'software': 'itom', 'lens': 'zoom'}
{'software': 'itom', 'institute': 'ITO', 'lens': 'zoom'}
{'software': 'itom', 'institute': 'none', 'lens': 'zoom'}
{'Institute': 'another', 'software': 'itom', 'institute': 'none', 'lens': 'zoom'}

```

Dictionaries aren't just for strings. Dictionary values can be any datatype, including strings, integers, objects, or even other dictionaries. Within a single dictionary, the values don't all need to be the same type; you can mix and match as needed. Dictionary keys are more restricted, but they can be strings, integers, and a few other types. You can also mix and match key datatypes within a dictionary.

```

1 d = {'lens': 'zoom', 'institute': 'ITO', 'software': 'itom'}
2 print(d)
3
4 d["version"] = 3
5 print(d)
6
7 d[42] = "douglas"
8 print(d)

```

```
{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 'version': 3}
{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 42: 'douglas', 'version': 3}
```

Deleting Items From Dictionaries

```
1 d = {'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 42: 'douglas', 'retrycount': 3}
2 print(d)
3
4 # delete individual items from a dictionary by key
5 del d[42]
6 print(d)
7
8 # delete all items from a dictionary
9 # Note that the set of empty curly braces in the output signifies a dictionary without any items.
10 d.clear()
11 print(d)
```

```
{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 42: 'douglas', 'retrycount': 3}
{'lens': 'zoom', 'institute': 'ITO', 'software': 'itom', 'retrycount': 3}
{}
```

Lists

Lists are Python's workhorse datatype. Variables can be named anything, and Python keeps track of the datatype internally.

Defining Lists A list is an ordered set of elements enclosed in square brackets. It can be used like a zero-based array.

```
1 # A list of five elements is defined. Note that they retain their original order.
2 li = ["a", "b", "zoom", "z", "example"]
3 print(li)
4
5 # The first element of any non-empty list is always li[0]
6 print(li[0])
7
8 # The last element of this five-element list is li[4].
9 print(li[4])
```

```
['a', 'b', 'zoom', 'z', 'example']
a
example
```

Negative List Indices A negative index accesses elements from the end of the list counting backwards. The last element of any non-empty list is always `li[-1]`. If the negative index is confusing to you, think of it this way: `li[-n] == li[len(li) - n]`.

```
1 li = ["a", "b", "zoom", "z", "example"]
2 print(li)
3 print(li[-1])
4
5 # In this list, li[-3] == li[5 - 3] == li[2]
6 print(li[-3])
```

```
['a', 'b', 'zoom', 'z', 'example']
'example'
'zoom'
```

Slicing a List You can get a subset of a list, called a *slice* by specifying two indices. The return value is a new list containing all the elements of the list, in order, starting with the first slice index up to but not including the second slice index.

If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first element you want, and the second slice index specifies the first element you don't want. The return value is everything in between.

```

1 li = ["a", "b", "zoom", "z", "example"]
2 print(li)
3
4 # A slice of li[1] up to but not including li[3] will be created
5 print(li[1:3])
6
7 # Slicing works if one or both of the slice indices is negative.
8 print(li[1:-1])
9
10 # Lists are zero-based, so li[0:3] returns the first three elements of the list,
11 # starting at li[0], up to but not including li[3].
12 print(li[0:3])
13
14 # If the left slice index is 0, you can leave it out, and 0 is implied. So li[:3]
15 # is the same as li[0:3]
16 print(li[:3])
17
18 # Similarly, if the right slice index is the length of the list, you can leave it
19 # out. So li[3:] is the same as li[3:5], because this list has five elements.
20 print(li[3:])
21
22 # If both slice indices are left out, all elements of the list are included.
23 # li[:] is shorthand for making a complete copy of a list.
24 print(li[:])

```

```

['a', 'b', 'zoom', 'z', 'example']
['b', 'zoom']
['b', 'zoom', 'z']
['a', 'b', 'zoom']
['a', 'b', 'zoom']
['z', 'example']
['a', 'b', 'zoom', 'z', 'example']

```

Adding Elements to Lists

```

1 li = ["a", "b", "zoom", "z", "example"]
2 print(li)
3
4 # Single elements can be added to the end of the list with append
5 li.append("new")
6 print(li)
7
8 # insert inserts a single element into a list. The numeric argument is the index of the
9 # first element that gets bumped out of position.
10 li.insert(2, "new")
11 print(li)
12
13 # Lists can be concatenated with extend. Note that you do not call extend with multiple
14 # arguments; you call it with one argument, a list. In this case, that list has two elements.
15 li.extend(["two", "elements"])
16 print(li)

```

```

['a', 'b', 'zoom', 'z', 'example']
['a', 'b', 'zoom', 'z', 'example', 'new']
['a', 'b', 'new', 'zoom', 'z', 'example', 'new']

```

```
['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
```

Difference between append und extend Lists have two methods, `extend()` and `append()`, that look like they do the same thing, but are in fact completely different. `extend()` takes a single argument, which is always a list, and adds each of the elements of that list to the original list. On the other hand, `append()` takes one argument, which can be any data type, and simply adds it to the end of the list.

```
1 # extend method
2 li = ['a', 'b', 'c']
3
4 # li is extended with a list of another three elements ('d', 'e', and 'f'), so you
5 # now have a list of six elements.
6 li.extend(['d', 'e', 'f'])
7 print(li)
8 print(len(li))
9 print(li[-1])
10
11 #append method
12 li = ['a', 'b', 'c']
13
14 # append method is called with a single argument, which is a list of three elements
15 # Now the list contains four elements because the last element appended is itself a
16 # list. Lists can contain any type of data, including other lists.
17 li.append(['d', 'e', 'f'])
18 print(li)
19 print(len(li))
20 print(li[-1])
```

```
['a', 'b', 'c', 'd', 'e', 'f']
6
f
['a', 'b', 'c', ['d', 'e', 'f']]
4
['d', 'e', 'f']
```

Searching Lists

```
1 li = ['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
2 print(li)
3
4 # index finds (only) the first occurrence of a value in the list and returns the index
5 print(li.index("example"))
6 print(li.index("new"))
7
8 # If the value is not found in the list, Python raises an exception.
9 # To test whether a value is in the list, use in, which returns True if the value is
10 # found or False if it is not.
11 print(li.index("c"))
```

```
5
2
Traceback (most recent call last):
  File "...", line ?, in ?
ValueError: 'c' is not in list
```

Deleting List Elements

```
1 li = ['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
2 print(li)
3
4 # remove (only) the first occurrence of a value from a list.
```



```

5 li.remove("z")
6 print(li)
7
8 # removes only the first occurrence of a value
9 li.remove("new")
10 print(li)
11
12 # pop removes the last element of the list, and it returns the value that it removed
13 li.pop()
14 print(li)
15
16 # If the value is not found in the list, Python raises an exception.
17 li.remove("c")

```

```

['a', 'b', 'new', 'zoom', 'z', 'example', 'new', 'two', 'elements']
['a', 'b', 'new', 'zoom', 'example', 'new', 'two', 'elements']
['a', 'b', 'zoom', 'example', 'new', 'two', 'elements']
['a', 'b', 'zoom', 'example', 'new', 'two']
Traceback (most recent call last):
  File "...", line ?, in ?
ValueError: list.remove(x): x not in list

```

Using List Operators Lists can also be concatenated with the + operator. `list = list + otherlist` has the same result as `list.extend(otherlist)`. But the + operator returns a new (concatenated) list as a value, whereas `extend` only alters an existing list. This means that `extend` is faster, especially for large lists.

```

1 li = ['a', 'b', 'zoom']
2 li = li + ['example', 'new']
3 print(li)
4
5 # li += ['two'] is equivalent to li.extend(['two'])
6 li += ['two']
7 print(li)
8
9 # The * operator works on lists as a repeater: li = [1, 2] * 3 is equivalent to
10 # li = [1, 2] + [1, 2] + [1, 2], which concatenates three lists into one.
11 li = [1, 2] * 3
12 print(li)

```

```

['a', 'b', 'zoom', 'example', 'new']
['a', 'b', 'zoom', 'example', 'new', 'two']
[1, 2, 1, 2, 1, 2]

```

Tuples

A tuple is an immutable list and can not be changed in any way once it is created. A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets. The elements of a tuple have a defined order and the indices are zero-based, just like a list.

Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list. Tuples can be converted into lists, and vice-versa. The built-in `tuple` function takes a list and returns a tuple with the same elements, and the `list` function takes a tuple and returns a list.

```

1 t = ("a", "b", "mpilgrim", "z", "example")
2 print(t)
3
4 # The first element of a non-empty tuple is always t[0]
5 print(t[0])
6

```

```

7  # Negative indices count from the end of the tuple, just as with a list.
8  print(t[-1])
9
10 # Slicing works too, just like a list.
11 print(t[1:3])

```

```

('a', 'b', 'mpilgrim', 'z', 'example')
a
example
('b', 'mpilgrim')

```

Keep in mind that tuples have not methods.

```

1  t = ('a', 'b', 'mpilgrim', 'z', 'example')
2
3  # all following examples cause errors
4  # t.append("new")
5  # t.remove("z")
6  # t.index("example")
7
8  # You can use "in" to see if an element exists in the tuple.
9  "z" in t

```

```
True
```

İ>ç.

If and else statements

if Statements

Perhaps the most well-known statement type is the *if* statement. For example:

```

1  x = int(input("Please enter an integer: "))
2
3  if x < 0:
4      print('Negative')
5  elif x == 0:
6      print('Zero')
7  else:
8      print('Positive')

```

There can be zero or more *elif* parts, and the *else* part is optional. The keyword ‘*elif*’ is short for ‘else if’, and is useful to avoid excessive indentation. An *if ... elif ... elif ...* sequence is a substitute for the *switch* or *case* statements found in other languages.

İ>ç.

Loops and Lists

for Statements

The *for* statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers, or giving the user the ability to define both the iteration step and halting condition, Python’s *for* statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```

1  # Measure some strings:
2  words = ['cat', 'window', 'defenestrate']
3

```

```

4 for w in words:
5     print(w, len(w))

```

```

cat 3
window 6
defenestrate 12

```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```

1 for w in words[:]:      # Loop over a slice copy of the entire list.
2     if len(w) > 6:
3         words.insert(0, w)
4 words

```

```
['defenestrate', 'cat', 'window', 'defenestrate']
```

The range Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```

1 for i in range(5):
2     print(i)

```

```

0
1
2
3
4

```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```

1 range(5, 10)           # 5 through 9
2 range(0, 10, 3)        # 0, 3, 6, 9
3 range(-10, -100, -30)  # -10, -40, -70

```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```

1 a = ['Mary', 'had', 'a', 'little', 'lamb']
2 for i in range(len(a)):
3     print(i, a[i])

```

```

0 Mary
1 had
2 a
3 little
4 lamb

```

In most such cases, however, it is convenient to use the `enumerate()` function.

break and *continue* Statements, and *else* Clauses on Loops

The *break* statement, like in C, breaks out of the smallest enclosing *for* or *while* loop.

Loop statements may have an *else* clause; it is executed when the loop terminates through exhaustion of the list (with *for*) or when the condition becomes false (with *while*), but not when the loop is terminated by a *break* statement. This is exemplified by the following loop, which searches for prime numbers:

```
1 for n in range(2, 10):
2     for x in range(2, n):
3         if n % x == 0:
4             print(n, 'equals', x, '*', n//x)
5             break
6         else:
7             # loop fell through without finding a factor
8             print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

When used with a loop, the `else` clause has more in common with the `else` clause of a *try* statement than it does that of *if* statements: a *try* statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the *try* statement and exceptions.

The *continue* statement continues with the next iteration of the loop:

```
1 for num in range(2, 10):
2     if num % 2 == 0:
3         print("Found an even number", num)
4         continue
5     print("Found a number", num)
```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

¶.

Classes and Objects

Classes

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below [Private Variables](#)), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

A Word About Names and Objects Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

Python Scopes and Namespaces Before introducing classes, I first have to tell you something about Python’s scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what’s going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let’s begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! ¹

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names

¹ Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module’s namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variable are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn't change *scope_test*'s binding of *spam*. The *nonlocal* assignment changed *scope_test*'s binding of *spam*, and the *global* assignment changed the module-level binding.

You can also see that there was no previous binding for *spam* before the *global* assignment. Classes introduce a little bit of new syntax, three new object types, and some new semantics.

Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions (*def* statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an *if* statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (*ClassName* in the example).

Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: *obj.name*. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

then *MyClass.i* and *MyClass.f* are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of *MyClass.i* by assignment. *__doc__* is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable *x*.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called `append`, `insert`, `remove`, `sort`, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string ‘hello world’. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:


```
xf = x.f
while True:
    print(xf())
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s object before the first argument.

If you still don’t understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn’t a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list. Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader

of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`. Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively *virtual*.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see <http://www.python.org/download/releases/2.3/mro/>. “Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly. Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method. User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```
raise Class

raise Instance
```

In the first form, `Class` must be an instance of `type` or of a class derived from it. The first form is a shorthand for:

```
raise Class()
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an `except` clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the `except` clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching `except` clause is triggered.

When an error message is printed for an unhandled exception, the exception’s class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`. By now you have

probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line)
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the

`yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left-off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function. Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
>>> sum(i*i for i in range(10))           # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))   # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

ï»¿.

Functions

Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
```

```

...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

```

The keyword *def* introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or *docstring*. (More about docstrings can be found in the section [Documentation Strings](#).) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a *global* statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [\[#\]](#) When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```

>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89

```

Coming from other languages, you might object that *fib* is not a function but a procedure since it doesn’t return a value. In fact, even functions without a *return* statement do return a value, albeit a rather boring one. This value is called *None* (it’s a built-in name). Writing the value *None* is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using *print()*:

```

>>> fib(0)
>>> print(fib(0))
None

```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```

>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that ‘belongs’ to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see [Classes](#)) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

More on Defining Functions It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print(complaint)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L
```



```
print(f(1))
print(f(2))
print(f(3))
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Keyword Arguments

Functions can also be called using **keyword arguments** of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')   # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)   # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
parrot(110, voltage=220)                 # duplicate value for the same argument
parrot(actor='John Cleese')              # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see [Mapping Types — dict](#)) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing

the positional arguments beyond the formal parameter list. (*name must occur before **name.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start*

and *stop* arguments. If they are not available separately, write the function call with the ***-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))          # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the ****-operator:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

Lambda Forms

By popular demand, a few features commonly found in functional programming languages like Lisp have been added to Python. With the `lambda` keyword, small anonymous functions can be created. Here's a function that returns the sum of its two arguments: `lambda a, b: a+b`. Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda forms can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

İ>ç.

Modules

Import Modules

Reload Modules in itom

see [here](#) to get more information about reloading modules in **itom**.

path Variablen

itom-packages Folder

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using the script editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

More on Modules A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module is imported somewhere.²

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use `imp.reload()`, e.g. `import imp; imp.reload(modulename)`.

² In fact function definitions are also 'statements' that are 'executed'; the execution of a module-level function enters the function name in the module's global symbol table.

Executing modules as scripts

When you run a Python module by executing the corresponding python script the code in the module will be executed, just as if you imported it, but with the variable `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

The Module Search Path

When a module named **spam** is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named *spam.py* in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- the directory containing the input script (or the current directory).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- the installation-dependent default.

After initialization, Python programs can modify `sys.path`:

The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section [Standard Modules](#) for more information.

Per default, the directory **itom-packages** (sub-directory of **itom** installation directory, is always part of the `sys.path` variable.

“Compiled” Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called *spam.pyc* exists in the directory where *spam.py* is found, this is assumed to contain an already-“byte-compiled” version of the module **spam**. The modification time of the version of *spam.py* used to create *spam.pyc* is recorded in *spam.pyc*, and the *.pyc* file is ignored if these don’t match.

Normally, you don’t need to do anything to create the *spam.pyc* file. Whenever *spam.py* is successfully compiled, an attempt is made to write the compiled version to *spam.pyc*. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting *spam.pyc* file will be recognized as invalid and thus ignored later. The contents of the *spam.pyc* file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some tips for experts:

- When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in *.pyo* files. The optimizer currently doesn’t help much; it only removes `assert` statements. When `-O` is used, *all* *bytecode* is optimized; *.pyc* files are ignored and *.py* files are compiled to optimized *bytecode*.

- Passing two `-O` flags to the Python interpreter (`-OO`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you’re doing.
- A program doesn’t run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that’s faster about `.pyc` or `.pyo` files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.
- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-O` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '__getframe__', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'callstats', 'copyright',
'displayhook', 'exc_info', 'excepthook',
'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version', 'version_info', 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Buffer
Error', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'Environme
ntError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'Generato
rExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexErr
or', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'P
endingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', '
StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'Ta
bError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'Unicod
eEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserW
arning', 'ValueError', 'Warning', 'ZeroDivisionError', '__build_class__', '__deb
ug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any',
'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'compile', '
complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate
', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memory
view', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property'
, 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sort
ed', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

sound/	Top-level package
__init__.py	Initialize the sound package
formats/	Subpackage for file format conversions
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	Subpackage for sound effects
__init__.py	
echo.py	


```

        surround.py
        reverse.py
        ...
filters/                               Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors

may also decide not to support it, if they don't see a use for importing `*` from their package. For example, the file `sounds/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound` package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from . import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practise in production code.

Remember, there is nothing wrong with using `from Package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

Intra-package References

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of `import` statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

ï»¿.

Prompting and passing

İ»¿.

Reading and writing files

Pickle -> load / dump

The following resources have been used:

- [Python-Kurs \(v3\) \(German\)](#)
- [Python Course \(v3\) \(English\)](#)
- [Official Python 3 documentation](#)
- [Moving from Python 2 to Python 3 \(Cheatsheet\)](#)
- [Dive into Python \(v3\)](#)
- [Learn Python The Hard Way \(be careful: Python 2\)](#)
- [Python Tutorial on javas2s.com \(v2\)](#)
- [Python examples \(example source code\) on javas2s.com \(v2\)](#)
- [Python Tutorial on tutorialspoint.com \(v2\)](#)

If you use **Python** in **itom** the following documents might give you more information about limitations, automatically reloading modules...:

9.1.2 Python Limitations in itom

You can use almost all methods, classes or modules provided by the core of **Python** or any external modules, if you consider the following hints or rules:

- The embedded python interpreter in **itom** is based on **Python 3.2**.
- **Python** is executed in its own thread in **itom**, therefore you should not use any modules that create some form of GUI. Usually the creation of any GUI element needs to be executed in the main thread of the calling application. Therefore don't use the python package **PyQt**. Instead use the GUI-functionality which is directly provided by **itom** and allows to extend the GUI of **itom**.
- If you were used to use to python methods, that asked the user for some input in the original python command window, you cannot use these commands in **itom**. Instead use message or input boxes that are provided by the class **ui** of the module **itom** in order to get some basic information from the user.
- Methods from the **threading** module of python are not fully usable in **itom**, since the interpreter lock, responsible for pseudo-threading in python, is not activated in **itom** yet.

9.1.3 Python - common problems and solutions

The following list state some common problems or known issues concerning **Python** (partially in combination with **itom**):

Re-Assigning a variable

Problem: If you assign an object with a limited access to anything (like a single camera) to a variable in a script, an access error might occur if you try to re-run the script:

```
cam = dataIO("IDSuEye", camera_id = 0)
```

Once you re-execute the same command, an error like the following one might be raised:

```
RuntimeError: Could not load plugin IDSuEye with error message:
Camera (0) could not be opened
```

This is due to the fact, that the re-execution at first tries to create a new instance of class *dataIO* with the same camera than is already opened. Then, this instance is assigned to the existing variable *cam*. In this moment, the recent content of *cam* is not longer in use and hence destroyed.

Solution: At first assign *None* or any other low-level value to *cam* and then re-assign the object that requires access to a limited structure. In some rare cases this is even not enough, since **Python** uses the concept of a garbage collector. Therefore, an object is only marked for deleted if it is not longer in use. The garbage collector is regularly called and finally deletes all marked objects. In this case, force the garbage collector to be executed:

```
import gc #import garbage collector

cam = None
gc.collect() #start the garbage collector
cam = dataIO("IDSuEye", camera_id = 0)
```

Variable deleted but referenced object is not closed

Problem: I delete a variable in **Python** but the value (e.g. a hardware instance - *dataIO* or *actuator*) is not closed.

Solution: At first, you should check if the variable you deleted is really the last variable that referenced to the underlying value. If you opened a hardware instance by the GUI you need to know that the GUI also holds a reference to the hardware. Therefore, the hardware must additionally be closed via a mouse click in the GUI. If the value is nevertheless not immediately destroyed, the last raised exception or the garbage collector of **Python** can be the reason.

```
class MyClass():
    def __init__(self):
        pass

    def __del__(self):
        print("MyClass destroyed")

m = MyClass()
raise RuntimeError(m)
del m
```

In this example, an instance of class *MyClass* is created (variable *m*). Afterwards, a runtime error is raised with *m* as single argument. Finally, *m* is deleted, but the destructor of *MyClass* is not called (no text *MyClass destroyed* is printed out). However, if you raise another runtime error:

```
raise RuntimeError()
```

the class is destroyed and the text appears. This is due to the fact, that the last exception that has been raised is still in memory and holds a reference to the passed argument, here, the instance *m* of class *MyClass*.

Note:

This behaviour changes in itom version > 1.4.0. Then, the last exception is not stored any more in the variables `sys.last_type`,

`sys.last_value` and `sys.last_traceback`.

Nevertheless, it might happen, that the object referenced by a variable (like a camera) is not immediately destroyed even if the last referencing variable is deleted. In **Python** objects are not directly deleted if they are not used any more, but they are only marked for deletion. Then, regularly, a garbage collector is executed that finally deletes all values marked for deletion. The reason is that deleting objects might be complicated and it is therefore better to execute this if the interpreter is idle or many objects have been marked. In order to directly force the garbage collector to delete marked objects, use:

```
import gc #import garbage collector
gc.collect() #start the garbage collector
```

Codec error

Problem: When executing a **Python** script, a syntax error with an error message similar to the following one appears:

```
File "C:\test.py", line 10
SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xe4 in position 0: unexpected end o
```

Solution: You used any special character (even in comments) in your code. Per default and if not otherwise state, a document is always parsed using the 'utf-8' codec. This codec does not support special characters (like German 'Umlaute' or the greek letter Αμ). If you want to use such characters, you need to indicate the codec of your file, e.g. by adding:

```
# -- coding: iso-8859-15 --
```

at the first line of your script file (iso-8859-15 represents the Western Europe charset). For more information see <https://www.python.org/dev/peps/pep-0263/>.

9.1.4 Reload modified modules

Usually, script files (hence *modules*) that are imported by another script in python are pre-compiled and cached for a faster execution once the script is loaded or imported another time. These cached files are always stored in a sub-folder **__pycache__** (file suffix .pyc). The advantage of this feature is a faster code execution once the pre-compiled and cached file is available. On the other hand, this feature can bring some drawbacks during the development process if the content of modules or packages may change. Then, these changes will not become active.

There are different possibilities to force Python to reload such a changed module:

1. Restart **itom**: The date of creation of all cached files (pyc) is compared with the change date of the corresponding py-files and they are recompiled if the scripts are newer.
2. The Python builtin-module **imp** provides mechanisms like the method **reload** to force Python to reload a specific module.
3. The mechanisms provided by the **imp** module are covered by the dialog **reload modules...** that is available in the menu **Script >> reload modules** of the main window of **itom**.
4. **itom** consists of a powerful auto-reload tool that can check all modules are their dependencies whether they have changed since the last check and reloads them. This tool is discussed in the course of this section.

At first, let us denote several issues that may happen due to the caching mechanism of **itom**. Consider the following three script files:

```
#script1.py
import mod2

print("version 1")
mod2.func2()
```

```
#mod2.py
import mod3

def func2():
    print("func2, version 1")
    mod3.func3()
```

```
#mod3.py

def func3():
    print("func3, version 1")
```

If *script1.py* is now executed, the modules *mod2* and *mod3* are imported and cached. The output at the first run is:

```
'version 1'
'func2, version 1'
'func3, version 1'
```

If we change now the strings ‘version 1’ to ‘version 2’ in all three files and execute *script1.py* again, the output will be as follows:

```
'version 2'
'func2, version 1'
'func3, version 1'
```

Only the first line changed, the other two stayed unchanged since *mod2* and *mod3* are still cached. Of course a restart of **itom** would lead to the right result:

```
'version 2'
'func2, version 2'
'func3, version 2'
```

Another possibility would be to reload *mod2* using the Python builtin-module *imp*, since *mod2* is imported by *script1.py*:

```
import imp
imp.reload(mod2)
```

Another execution of *script1.py* will now lead to the following result:

```
'version 2'
'func2, version 2'
'func3, version 1'
```

The last result, coming from *mod3* is still unchanged. This comes due to the fact that the *imp.reload* command does not resolve any dependencies but only tries to reload one single module, corresponding to the content of one single py-file. Therefore, you always need to know where exactly code changes have occurred and reload all related modules. To simplify this mechanism, you can use the dialog **Reload modules...** that is reachable via the **itom** menu **Script >> reload modules >> reload modules...**. Sometimes, the reload may fail. Reasons for this and further limitations of the reload process are discussed later.

In order to provide an easy way to automatically reload all modules that have been changed since the last execution, **itom** provides an auto-reload tool. This tool has been inspired by the autoreload module of *IPython* (<http://ipython.org/ipython-doc/dev/config/extensions/autoreload.html>) and is fully integrated into **itom**. Enable the tool by the menu **Script >> reload modules >> autoreload modules**. Depending on further settings, the currently executed script file, code command or function is checked (including all its dependencies) for changes are reloaded if necessary. You have full control in which cases you want that check being executed. This is controlled by the further options in the submenu **Script >> reload modules**:

- autoreload before script execution: The check is executed whenever you run or debug a script file
- autoreload before single command: The check is executed before you execute a string command from the command line of **itom**
- autoreload before events and function calls: The check is executed if any python code or function is executed due to an event or signal (e.g. button click in a GUI)

Try to enable the autoreload tool and enable at least the option *autoreload before script execution*. Then change the version strings in all files to ‘version 3’ and execute *script3*:

```
'version 3'
'func2, version 3'
'func3, version 3'
```

Using this tool, you do not need to worry about reloading any changed modules. This gives you a powerful tool for developing more complex scripts that are divided into multiple files. The autoreload tool can also be enabled and configured using the command `itom.autoReloader()`.

Sometimes, you will notice that reloading a module using the *imp* module will fail or not work. Consider the following script:

```
#mod4.py

class MyRect():
    def __init__(self, height, width):
        self.sizes = [height, width]

    def getSizes(self):
        print("size of MyRect", self.sizes)
```

Now type into the command line:

```
import mod4
rect = mod4.MyRect(4,5)
rect.getSizes()
```

You will obtain:

```
'size of MyRect: [4,5]'
```

If you change now the print-command in the method `getSizes` of class `MyRect` to:

```
print("width:", self.sizes[1], ",height:", self.sizes[0])
```

and call:

```
imp.reload(mod4)
```

in order to reload *mod4.py* again, a call to:

```
rect.getSizes()
```

in the command line will still lead to the old result. This is due to the fact, that the *imp* module cannot reload objects that are still referenced by another variable. In this case, the global variable `rect` is an instance of the class `MyRect`. Therefore, it is not possible to reload this class before deleting the variable `rect`. However, if you enable the autoreload tool and enable the option **autoreload before single command** before changing the print command, you will see that this still is also able to replace the code of a class method even if there are already active instances of this class.

The autoreload tool is a way more powerful than the native *imp* implementation. However there are still some limitations:

Reloading Python modules in a reliable way is in general difficult, and unexpected things may occur. 'autoreload' tries to work around common pitfalls by replacing function code objects and parts of classes previously in the module with new versions. This makes the following things to work:

- Functions and classes imported via `'from xxx import foo'` are upgraded to new versions when `'xxx'` is reloaded.
- Methods and properties of classes are upgraded on reload, so that calling `'c.foo()'` on an object `'c'` created before the reload causes the new code for `'foo'` to be executed.

Some of the known remaining caveats are:

- Replacing code objects does not always succeed: changing a `@property` in a class to an ordinary method or a method to a member variable can cause problems (but in old objects only).

- Functions that are removed (eg. via monkey-patching) from a module before it is reloaded are not upgraded.
- C extension modules cannot be reloaded, and so cannot be autoreloaded.

(taken from <http://ipython.org/ipython-doc/dev/config/extensions/autoreload.html>)

9.2 Python module itom

The main purpose of the embedded **Python** interpreter in **itom** is to access the specific functionalities provided by **itom**. This is done by the **Python**-module *itom*, that is importable only in the embedded **Python** interpreter in **itom**. This module includes interfaces for hardware and algorithm plugins of **itom** as well as classes that wrap the most important internal data structures of **itom**, like matrices (class *dataObject*), point clouds (class *pointCloud*) or polygon meshes (class *polygonMesh*). Additionally the module provides functions to manipulate or extend the graphical user interface of **itom** as well as to create own dialogs or windows (provided by the class *ui* and *uiItem*).

More information about the module *itom* can be found under:

9.2.1 Python-Module itom

The module *itom* is a python module, that builds the connecting part between the embedded python script language and the overall **itom** application. This module is only available in the context of a running **itom** software.

Import itom

Like any other python module, you need to import the module *itom* in your script. Usually, the import is done at the beginning of a script. In the following example, it is shown how the method **filterHelp** of the module *itom* can be called with respect to different ways of import *itom*:

```
1  #1. import the whole module. Any method is then directly accessible
2  from itom import *
3  filterHelp("...")
4
5  #2. import the itom module without the global access to its methods
6  import itom
7  itom.filterHelp("...")
8
9  #3. only import certain methods or classes of the module itom
10 from itom import filterHelp as anyAlias
11 anyAlias("...")
```

If you simply type any commands in the **itom** command line or if you directly execute any script at the top level, you don't need import the module *itom*, since this module already has been imported at startup of **itom** with

```
from itom import *
```

However, if you import another script file in your main script file and you want to access any methods of *itom* in this secondary script, then you need to import *itom* in this script using one of the methods shown above.

Content of itom

- class **dataObject**. This is the **itom** internal matrix class, compatible to *Numpy*, and also used by any connected grabber or camera. Every matrix is extended by fixed and user-defined tags and keywords. For an introduction to the data object, see *DataObject*, a full reference is available under *dataObject*. The data object is compatible to any numpy array, however the tags and keywords will get lost.

- class **npDataObject**. This class is inherited from **numpy.array** and extends it by the tags and keywords, also available in the data object. For more information see *npDataObject* or the script reference *npDataObject*.
- classes **ui** and **uiItem** are the main classes for creating user defined dialogs and windows in **itom** and show them using some lines of script code. For more information about their use, see *Creating advanced dialogs and windows* or their definitions in the script reference *ui* and *uiItem*.
- class **dataIO** is the class in order to access any plugin instance of type **dataIO** (cameras, grabbers, AD-converter...). The full reference can be found under *dataIO*.
- class **actuator** is the class in order to access any plugin instance of type **actuator**, like motor stages... The full reference can be found under *actuator*.
- **other class free methods.** *itom* also directly contains a lot of methods, that makes features of **itom** accessible by a Python script:
 - add or remove buttons or items to the **itom** menu and toolbar (see *Customize the menu and toolbars of itom*)
 - get help about plugins and their functionality
 - call any algorithm or filter, provided by a plugin of type **algo**
 - directly plot matrices like *dataObjects*.

Contents:

DataObject

Introduction In **itom**, the class *dataObject* is the main array object. Arrays in **itom** can have the following properties:

- unlimited number of dimensions
- each dimension can have an arbitrary size
- **possible data types:**

```
"uint8"      #unsigned integer, 8 bit [0,255]
"int8"       #signed integer, 8 bit [-128,127]
"uint16"     #unsigned integer, 16 bit [0,65536]
"int16"      #signed integer, 16 bit [-32768,32767]
"uint32"     #unsigned integer, 32 bit
"int32"      #signed integer, 32 bit
"float32"    #floating point, 32 bit single precision
"float64"    #floating point, 64 bit double precision
"complex64"  #complex number with two float32 components
"complex128" #complex number with two float64 components
```

Before giving a short tutorial about how to use the class *dataObject*, the base idea and concept of the array structure should be explained. If you already know the huge **Python** module **Numpy** with its base array class **numpy.array**, one will ask why another similar array class is provided by **itom**. The reasons for this are as follows:

- The python class *dataObject* is just a wrapper for the **itom** internal class **DataObject**, written in C++. This array structure is used all over **itom** and also passed to any plugin instances of **itom**. Internally, the C++ class **DataObject** is based on OpenCV-matrices, such that functionalities provided by the open-source Computer-Vision Library (OpenCV) can be used by **itom**.
- The class **dataObject** should also be used to store real measurement data. Therefore it is possible to add tags and other meta information to every *dataObject* (like axes descriptions, scale and offset values, protocol entries...).
- Usually, array classes (like the class **Numpy.array**) store the whole matrix in one non-interrupting block in memory. Due to the working principle of every operating system, it is sometimes difficult to allocate

a huge block in memory. Therefore, **dataObject** only stores the sub-matrices of the last two-dimensions in single blocks in memory, while the first **n-2** dimensions of the array are represented by one vector in memory, where every cell is pointing to the corresponding sub-matrix (called plane). Using this concept, huger arrays can be allocated without causing a memory error.

Creating a dataObject In general, a *dataObject* is created like any other class instance in **Python**, hence the constructor of class *dataObject* is called. For a full reference of the constructor of class **dataObject**, type

```
help(dataObject)
```

In the following example, some dataObjects of different size and types are created. Using these constructors, the content of the created array is arbitrary at initialization:

```
1 #1. empty dataObject, dimensions: 0, size: []
2 a = dataObject()
3
4 #2. one dimensional dataObject
5 # a one dimensional dataObject already is
6 # allocated as an array of size [1 x n]
7 b = dataObject([5], "float32") #size [1x5]
8
9 #3. 5 x 3 array, type: int8
10 c = dataObject([5,3], "int8")
11
12 #4. 2 x 5 x 10 array, type: complex128
13 # here two planes of size [5x10] are created and a vector with two items points to them
14 d = dataObject([2,5,10], "complex128")
15
16 #5. 2 x 5 x 10 array, type: complex128, continuous
17 # This matrix has the same size and type than matrix
18 # 'd' above. However, the continuous keyword indicates,
19 # that python should already allocate all planes in
20 # one block. Then the data object can be converted in
21 # a numpy.array without the need of copying the data block
22 # in memory. It is useful to use this keyword, if you
23 # often want to switch between dataObject and numpy.arrays.
24 # However consider that this is not recommended for huge
25 # matrices.
26 e = dataObject([2,5,10], "complex128", continuous = True)
```

You can also use the copy constructor of class **dataObject** in order to create a dataObject from another array-like object or a sequence of numbers (tuple, list...). In **Python** it is usual, that different objects share their memory (for arrays the memory is mainly the data block(s)) as long as possible, such that memory and execution time is saved. This is also the case when using the copy constructor. See the **Numpy** documentation for more information about this. The main thing you should know is, that if you change the value of any cell of an array, the corresponding value is also changed in all arrays, that share their memory with the dataObject.

```
1 #1. create dataObject from any array-like object (e.g. Numpy array)
2 import numpy as np
3 a = np.ndarray([5,7])
4 b = dataObject(a) #b has the continuous flag set
5
6 #2. create dataObject from a tuple of values
7 # any object, that python can interpret as sequence can be used
8 # in order to initialize the data object. The dataObject can have
9 # an arbitrary size or number of dimensions, if the total number
10 # of elements fits to the length of the given input sequence.
11 # In this case, the sequence is totally copied into the data object.
12 # The values are filled row-by-row into the array, also called as
13 # c-continuous creation.
14 c = (2,7,4,3,8,9,6,2) #8 values
15 d = dataObject([2,4], data = c)
```

```

16 #3. create a dataObject as shallow copy of another dataObject
17 e = dataObject(d)
18

```

Some text about the basic idea of the `dataObject` and how it works. Some additional pictures.

For a detailed methods-summary of the `dataObject` see [itom Script Reference](#).

npDataObject

Some text about the basic idea of the `npDataObject` and how it works. Some additional pictures.

For a detailed methods-summary of the `npDataObject` see [itom Script Reference](#).

Plugin-Interface and Plugins

The basic plugin idea again, some more details,

a description of type, initialisation, ... how to work with, how to delete them, how to get help,

For a detailed methods-summary of **dataIO**, **actuator**, **filter** and **widgets** see [itom Script Reference](#).

9.2.2 Load and save images and other files

Native file formats

itom has a native support to load and save various file formats. Additionally, algorithm plugins can provide further filters to load and save more file formats. All supported formats (native and plugin-based) are considered in the GUI such that you can open a file using the *file >> open* menu, by double-clicking a file in the file system dialog or by using the import / export buttons in the workspace.

The following formats are natively supported:

- **idc**: This is the default itom file format (itom data collection) and is able to store entire Python structures (e.g. dictionaries, lists, tuples...) containing data objects, point clouds or other python objects. This file format is written and read using the module `pickle` from python.
- **mat**: Similar to **idc** you can import or export entire data structures from and to Matlab. This is only available if the package **scipy** is installed.
- **ido**, **idh**: This is a xml-based data format for single data objects or only meta and header information of a single data object.

Here are some examples about the natively supported. At first, let us create some exemplary data objects:

```

#randomly filled matrix of size 100x100, type: uint8
obj1 = dataObject.randn([100,100], 'uint8')

#50x100 matrix filled with float32 values
obj2 = dataObject([50,100], 'float32')
obj2[0:25,:] = 0.0
obj2[25:50,:] = 10.3
obj2[25:50,50:100] = -5.25

#768x1024 coloured data object (type: rgba32), background: white,
#left side: transparent, in the middle three horizontal bars in red,
#green and blue.
obj3 = dataObject([768,1024], 'rgba32')
obj3[:,0:512] = rgba32(255,255,255,0) #transparent (alpha=0)
obj3[:,512:1024] = rgba32(255,255,255,255)
obj3[100:300,200:800] = rgba32(255,0,0,255) #red

```

```
obj3[300:500,200:800] = rgba32(0,255,0,255) #green
obj3[500:700,200:800] = rgba32(0,0,255,255) #blue
```

Now, we want to save all these three data objects together into one **idc** file using the method `itom.saveIDC()`. **idc** files always contain a dictionary where you can save whatever you want to (not only data objects). When loading an **idc** file (by `itom.loadIDC()`), you get the original dictionary back:

```
# save idc file
saveIDC("C:/test.idc", {"mat1":obj1, "mat2":obj2, "mat3":obj3})
# remember: if you use \ in pathes, replace them by \\

# load the file again
myDict = loadIDC("C:/test.idc")
obj1new = myDict["mat1"]
obj2new = myDict["mat2"]
```

Note: In these examples, the methods and classes from the `itom` are written without the module name as prefix. These is possible, since the `itom` is globally imported at startup of itom. However this holds only for the global workspace.

If you wish to save the same objects to a Matlab **mat** file, this is also possible via dictionaries. When loaded in Matlab, each item in the dictionary is a variable in the workspace whose name is the key of the item. The save and load methods are `itom.saveMatlabMat()` and `itom.loadMatlabMat()`:

```
# save matlab file
saveMatlabMat("C:/test.mat", {"mat1":obj1, "mat2":obj2, "mat3":obj3})

# load the file again
myDict = loadMatlabMat("C:/test.mat")
obj1new = myDict["mat1"]
obj2new = myDict["mat2"]
```

If a data object is saved in a Matlab **mat** file, Matlab will load this data object as cell array that contains both the matrix data itself and all meta information (scaling, offset, tags, ...).

If you want to export single data objects in a readable format, use the methods `itom.saveDataObject()` and `itom.loadDataObject()`. Both export or import into / from the xml-based files **ido** (entire data object with data and meta information) and **idh** (only meta information (header) of data object). In the first format, header information is directly readable in the file while the matrix data is encoded in a base64 format.

Plugin-based file formats

Plugins can provide filters for saving or loading the following objects:

- data objects
- point clouds
- polygon meshes

If any filter indicates to support the corresponding file input or file output interface, this filter is automatically recognized and integrated in the GUI. Nevertheless, these filters can be called like any other filter in **litoml**.

Most filters for loading any image formats are included in the plugin **dataObjectIO**. The filter documentation of this plugin gives detailed information about every single filter. Loading or saving point clouds or polygonal meshes are included in the plugin **PclTools**.

Image file formats

As mentioned in the section above, plugins can provide filters to save or load data objects. The plugin **dataObjectIO** contains many filters to save into common image formats and load them back to data objects. Click **info** in the context menu of any algorithm filter to get more information about this filter.

All image-based file filters follow these rules how to handle different data types:

- uint8 or uint16 are saved as gray-values (8bit or if supported as 16bit) or if the image format allows color are saved according to the defined color palette.
- float32 or float64 are saved as gray-values (8bit or if supported as 16bit) or according to the defined color palette. Therefore the values must be between 0.0 and 1.0. Values outside these borders are clipped. If the image format supports RGBA, invalid values are saved as transparent values (alpha=zero) else as black values.
- rgba32 can be saved as 'rgb' (full opacity), 'rgba' (alpha channel is considered, not supported by all formats) or gray formats, where the color image is transformed to gray. if a format from a color palette is indicated, the color image is transformed to gray first and then interpreted using the indicated color palette.

Among others, the following color formats are supported: bmp, jpg, png, gif (read-only), tiff, xpm, xbm, ras, pgm, ppm...

Loading these files can mainly be achieved by the filter **loadAnyImage**:

```
reload_tiff_rgba=dataObject()
filter("loadAnyImage", reload_tiff_rgba, 'pic_rgba.tiff', 'asIs')
```

'asIs' means that the data is loaded without further transformations (if possible), hence, a color data format is loaded to a rgba32 data object, a uint8 gray image is loaded to uint8 and so on. However, you can also choose that you want the image to be always converted to gray, you can choose a specific color channel...

For saving to different color formats, there is usually a specific filter for each format. This allows passing further individual parameters like the color map for *png*. This indicates if fixed- or floating-point data objects should be interpreted with a specific color map. The output is then a color image instead of a gray one:

```
filter("savePNG", obj1, 'C:/pic_falseColor.png', 'hotIron')
```

For more examples about saving and loading data, see the demo file **demoLoadSaveDataObjects.py** in the demo-folder.

9.2.3 Glossary

(non-)continuous data object

per default, matrixes in the data object are stored as follows: the last two dimensions are stored continuously in a two dimensional matrix by the help of openCV-Mat-structures. All these 2dim-matrixes are then stored in a one-dimensional vector, however the relation between the position in this vector and the first ($n-2$) dimensions, where n is the number of total dimensions, is given by a simple equation, which can for instance be found in the *openCV* documentation. Since the data blocks for each two-dimensional matrix must not be stored continuously in memory, this method is called *non-continuous data object*.

In order to realize a compatible version with respect to *numpy*, *matlab*... the data can also be stored *continuously*. The basic structure for the data object is the same than in the *non-continuous* (default) version, but the data of each 2dim-matrix lies continuously in memory and each data-pointer of each matrix just points to the first element of the corresponding matrix in this big data block in memory. Data objects of class *ndDataObject* are always organized as continuous data object, while ordinary data objects (class *dataObject*) can be continuous or non-continuous.

The non-continuous representation has advantages especially with respect to huge data sets, since it is more difficult to get a big continuous block in memory without reorganizing it than multiple smaller blocks of memory, which can be distributed randomly in memory.

Matrixes with only one or two dimension are automatically stored continuously.

Deep Copies

Are copies

Shallow Copies

are copied references

Python Version

The full script reference of the module `itom` can be found under *itom Script Reference*.

9.3 Further python packages

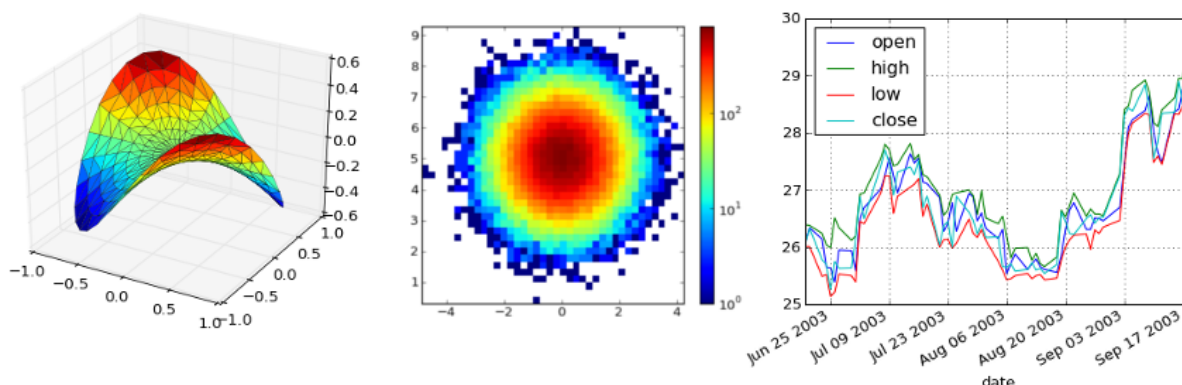
Beside the module `itom`, it is recommended to use the functionalities provided by the **Python** packages **Numpy**, **Scipy** and **Matplotlib**. During the development of `itom` a high compatibility especially to those modules has been taken into account. For instance it is possible to render the **Matplotlib** figures in user defined windows, created by the class `ui` of the module `itom` (see *Creating advanced dialogs and windows*). Additionally, the **Numpy** array is compatible to the `itom` internal `dataObject` or `npDataObject`.

9.3.1 Python-Module numpy

9.3.2 Python-Module scipy

9.3.3 Python-Module matplotlib

The module **matplotlib** can be used in order to create two or three dimensional plots like in these examples:



Since the internal plotting methods of `itom` mainly provide display widgets for plotting matrix contents and images, **matplotlib** can be used for plotting other types of graphics like graphs, line plots, bars... A huge list of examples can be found under <http://matplotlib.org/gallery.html>.

Set Matplotlib backend to itom

In order to render the output of **matplotlib** into an `itom` window or a user interface generated by `itom` (see *Creating advanced dialogs and windows*), you need to write the following command before importing any module of the package **matplotlib**:

```
import matplotlib
matplotlib.use('module://mpl_itom.backend_itomagg', False)
```

Alternatively, you can always configure **matplotlib** to render its output in `itom` windows. Therefore, `itom` internally has an environment variable **MPLCONFIGDIR** that points to the directory:

```
itom-packages/mpl_itom
```

of your `itom` installation (or build-folder if self-compiled). If you can place a copy of the matplotlib config file **matplotlibrc** into this directory and modify the variable **backend** to **module://mpl_itom.backend_itomagg**.

A template for the configuration file can be either found in the Python subfolder `[PythonDir]/Lib/site-packages/matplotlib/mpl-data` or under http://matplotlib.org/_static/matplotlibrc.

This is the part you need to change:

```
# the default backend; one of GTK GTKAgg GTKCairo GTK3Agg GTK3Cairo
# CocoaAgg MacOSX Qt4Agg TkAgg WX WXAgg Agg Cairo GDK PS PDF SVG
# Template
# You can also deploy your own backend outside of matplotlib by
# referring to the module name (which must be in the PYTHONPATH) as
# 'module://my_backend'
backend      : module://mpl_itom.backend_itomagg
```

Note: Once you placed the config file, you don't need to use the `use` command in any of your scripts.

If you are not sure, whether your user defined config file is loaded, you can obtain the path to the loaded config file with:

```
>>> import matplotlib
>>> matplotlib.matplotlib_fname()
```

For more information about this, see <http://matplotlib.org/users/customizing.html>

Simple Matplotlib example

This example shows you that is possible to use any arbitrary matplotlib python script and execute it in **itom**. Therefore, the example `hist2d_log_demo.py` from the pylab examples on <http://matplotlib.org> is taken.

Its source code is:

```
import matplotlib
matplotlib.use('module://mpl_itom.backend_itomagg', False)
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.collections import EllipseCollection

x = np.arange(10)
y = np.arange(15)
X, Y = np.meshgrid(x, y)

XY = np.hstack((X.ravel()[:, np.newaxis], Y.ravel()[:, np.newaxis]))

ww = X/10.0
hh = Y/15.0
aa = X*9

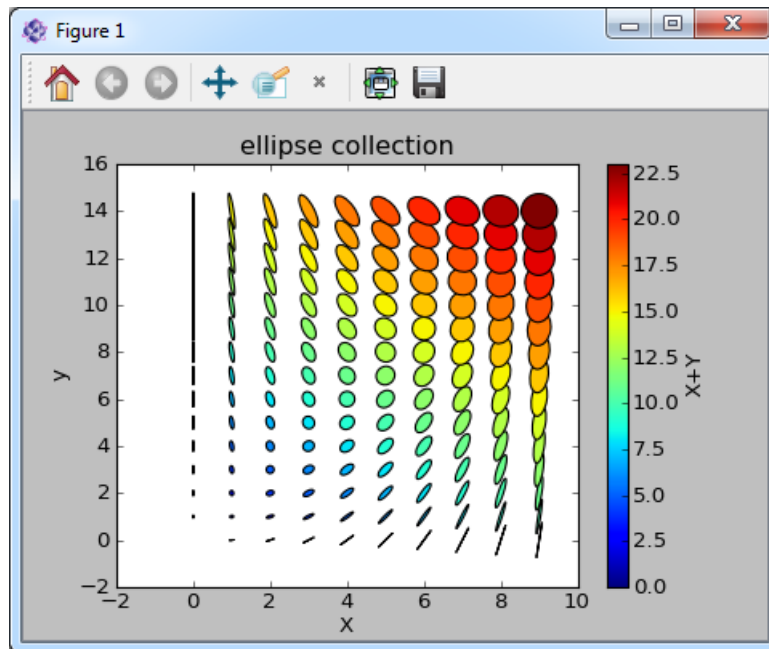
ax = plt.subplot(1, 1, 1)

ec = EllipseCollection(
    ww,
    hh,
    aa,
    units='x',
    offsets=XY,
    transOffset=ax.transData)
ec.set_array((X+Y).ravel())
ax.add_collection(ec)
ax.autoscale_view()
ax.set_xlabel('X')
ax.set_ylabel('Y')
cbar = plt.colorbar(ec)
cbar.set_label('X+Y')
```



```
title("ellipse collection")
plt.show()
```

Please consider that the original source code has been changed such that the first two lines are prepended. After executing this script, the following figure is displayed in **itom**:



Note: If the figure does not appear, the matplotlib designer widget for **itom** is not available. This means, the library **matplotlibPlot** in the **designer** folder of **itom** is missing.

Further examples from the official matplotlib gallery are contained in the itom subfolder **demo/matplotlib**.

Embedding a matplotlib figure in your own user interface

itom not only provides stand-alone windows for showing the result of the *matplotlib*, but it is also possible to integrate a *matplotlib* canvas into own user interfaces created by the QtDesigner and scripted with **Python**. For more information how to do this, see *Creating advanced dialogs and windows*.

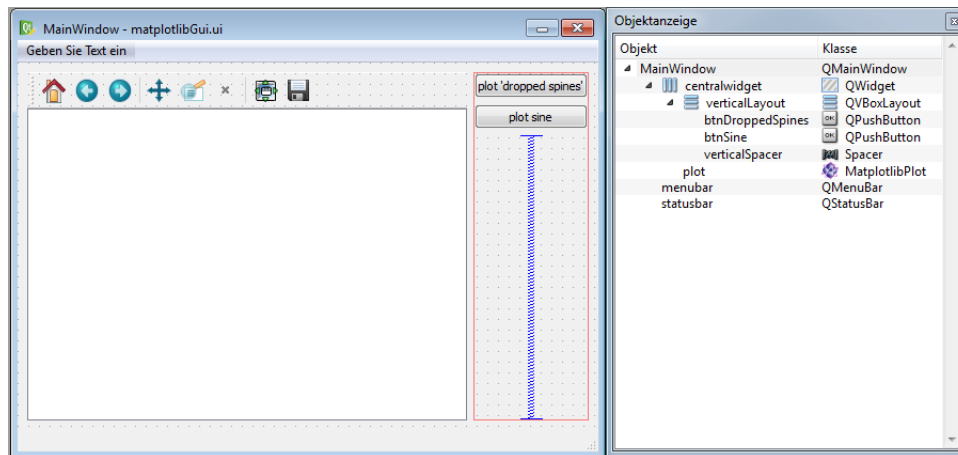
In the widget library of QtDesigner there is the widget **MatplotlibPlot** in the section **itom Plugins** (under the consumption that the corresponding designer plugin library is contained in the folder *designer* of the root directory of **itom**). Drag&Drop an instance of this widget onto your user interface.

In the following example, a new main window is created where a *MatplotlibPlot* widget (name: *plot*) is placed on the left side while two buttons (name: *btnDroppedSpines* and *btnSine*) are placed on the right side:

When any of the both buttons are pressed, the following example should be displayed in the figure **plot** on the left side.

```
import matplotlib
matplotlib.use('module://mpl_itom.backend_itomagg', False)
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def plotDroppedSpines():
    '''
    plot taken from matplotlib example 'spines_demo_dropped.py'
    '''
```

```

canvas = gui.plot #reference to matplotlibPlot widget
fig = plt.figure(num = 3, canvas=canvas)
ax = fig.add_subplot(111)
ax.clear()

image = np.random.uniform(size=(10, 10))
ax.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
ax.set_title('dropped spines')

# Move left and bottom spines outward by 10 points
ax.spines['left'].set_position(('outward', 10))
ax.spines['bottom'].set_position(('outward', 10))
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')

plt.show()

```

```

def plotSine():
    '''
    plots sine, taken from matplotlib gallery examples
    '''
    t = np.arange(0.0, 1.0, 0.01)
    s = np.sin(2*np.pi*t)

    canvas = gui.plot #reference to matplotlibPlot widget
    fig = plt.figure(num = 3, canvas=canvas)
    ax = fig.add_subplot(111)
    ax.clear()
    ax.plot(t,s)

    plt.show()

gui = ui("matplotlibGui.ui", type = ui.TYPESWINDOW)
gui.btnSine.connect("clicked()", plotSine)
gui.btnDroppedSpines.connect("clicked()", plotDroppedSpines)
gui.show()

```

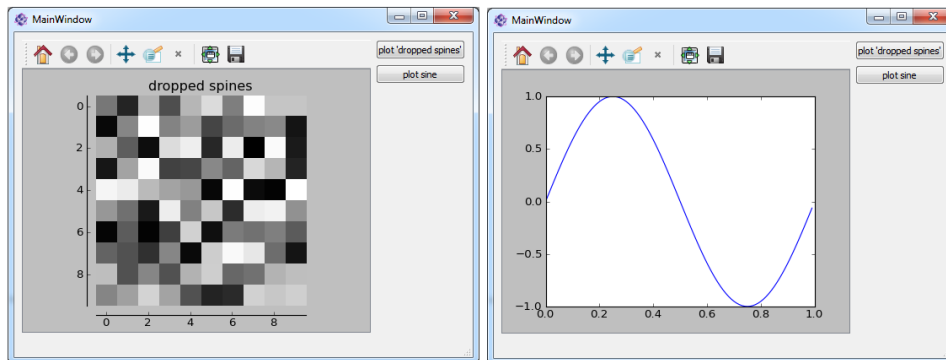
```

# if you call this script for the second time, the given figure-num (3)
# is already in used for the lastly closed figure. Therefore also tell
# matplotlib to close this figure handle.

```

```
plt.close(3)
```

The result is:



What happens here?

- At the end of the script, the user interface *matplotlibGui.ui* is loaded and referenced by the variable *gui*.
- The click-events of both buttons is connected to the methods *plotSine* and *plotDroppedSpines* respectively.
- The gui is shown

For both button clicks the following things have to be done:

Once you added the **itom**-backend command as first, mandatory line to your script, the *figure*-class of *matplotlib* has got one further keyword-based parameter *canvas*. This needs to be used in order to tell the figure where the widget is to plot the content to. If you omit this parameter, a new window is opened with the corresponding output. If you set this parameter to the reference of the widget of type *MatplotlibPlot* (here: called *canvas*), the output is print there.

The you have the reference to the figure-instance of *matplotlib* and can go one like usual.

Note: Once you created one figure that maps to a given widget using the *canvas*-keyword, this figure is not deleted when a new figure is created using the same keyword. Therefore it will happen that lots of invisible figures need to be handled. Therefore, the *num* keyword argument is used in the methods in the example in order to always tell *matplotlib* that a defined figure with the handle 3 should be instantiated. If this handle already exists, this existing figure is used. Therefore it is also necessary to clear the axes using **ax.clear()**.

Furthermore, if you created a figure with a given *num* and *canvas*, deletes the user interface and creates a new one, a new figure with the handle of the old one is not able to plot in the new user interface since it still is connected with the old, deleted widget. Therefore, the command:

```
plt.close(3)
```

is used to firstly delete the *matplotlib*-figure with handle 3 once the script is re-executed.

Note: Usually, *matplotlib* is allowed changing the size of the output window. The window is then forced to have a new size that can afterwards be manually resized. If your output widget is embedded in an user interface, this behaviour might be undesired. Then disable it by setting the property **forceWindowResize** to **False**. In the example above this can be done by:

```
gui.plot["forceWindowResize"] = False
```

or by directly setting the corresponding property when designing the user interface in QtDesigner.

This example is contained in the **demo/ui/embeddedMatplotlib** folder.

Other recommended packages are:

- **scikit-image** for image processing

- **PIL** is the python image library
- **sphinx** for creating this documentation

See *python package manager* for more information about getting packages.

9.4 Tutorials, documentations about Python 3

See the following sources for more information, tutorials and documentations about Python 3 and Numpy:

- [Python-Kurs \(v3\) \(German\)](#)
- [Python Course \(v3\) \(English\)](#)
- [Official Python 3 documentation](#)
- [Moving from Python 2 to Python 3 \(Cheatsheet\)](#)
- [Dive into Python \(v3\)](#)
- [Numpy for Matlab users](#)

ITOM SCRIPT REFERENCE

This reference gives a full reference for all script commands that are contained in the python module *itom*.

Content:

10.1 itom methods

10.1.1 Plotting and camera

`itom.liveImage (cam[, className, properties])` → show a camera live image in a newly created figure
Creates a plot-image (2D) and automatically grabs images into this window. This function is not blocking.

Parameters **cam** : {dataIO-Instance}

Camera grabber device from which images are acquired.

className : {str}, optional

class name of desired plot (if not indicated or if the className can not be found, the default plot will be used (see application settings))

properties : {dict}, optional

optional dictionary of properties that will be directly applied to the plot widget.

`itom.plot (data[, className, properties])` → plots a dataObject in a newly created figure
Plot an existing dataObject in dockable, not blocking window. The style of the plot depends on the object dimensions. If x-dim or y-dim are equal to 1, plot will be a line-plot, else a 2D-plot.

Parameters **data** : {DataObject}

Is the data object whose region of interest will be plotted.

className : {str}, optional

class name of desired plot (if not indicated or if the className can not be found, the default plot will be used (see application settings))

properties : {dict}, optional

optional dictionary of properties that will be directly applied to the plot widget.

10.1.2 Using algorithms and filters

`itom.filter (name[, furtherParameters, ...])` → invoke a filter (or algorithm) function from an algorithm-plugin.

This function is used to invoke itom filter-functions or algorithms, declared within itom-algorithm plugins. The parameters (arguments) depends on the specific filter function (see filterHelp(name)), By filterHelp() a list of available filter functions is retrieved.

Parameters `name` : {str}

The name of the filter

furtherParameters : {variant}

Further parameters depend on the filter-methods itself (give the mandatory and then optional parameters in their defined order).

Returns `out` : {variant}

The returned values depend on the definition of each filter. In general it is a tuple of all output parameters that are defined by the filter function.

See also:

[*filterHelp*](#)

10.1.3 Methods for getting help and information about filters, plugins, ui-elements, ...

`itom.pluginHelp(pluginName[, dictionary = False])` → generates an online help for the specified plugin.

Gets (also print to console) the initialisation parameters of the plugin specified `pluginName` (str, as specified in the plugin window). If `dictionary == True`, a dict with all plugin parameters is returned and nothing is printed to the console.

Parameters `pluginName` : {str}

is the fullname of a plugin as specified in the plugin window.

dictionary : {bool}, optional

if `dictionary == True`, function returns a dict with plugin parameters and does not print anything to the console (default: False)

Returns `out` : {None or dict}

Returns None or a dict depending on the value of parameter `dictionary`.

`itom.widgetHelp([widgetName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given widget(s).

This method prints information about one specific widget or a list of widgets to the console output. If one specific widget, defined in an algorithm plugin can be found that case-sensitively fits the given widget-Name its full documentation is printed. Else, a list of widgets is printed whose name contains the given widgetName.

Parameters `widgetName` : {str}, optional

is the fullname or a part of any widget-name which should be displayed. If `widgetName` is empty or no widget matches `widgetName` (case sensitive) a list with all suitable widgets is given.

dictionary : {dict}, optional

if `dictionary == 1`, a dictionary with all relevant components of the widget's documentation is returned and nothing is printed to the command line [default: 0]

furtherInfos : {int}, optional

Usually, widgets whose name only contains the given `widgetName` are only listed at the end of the information text. If this parameter is set to 1 [default: 0], the full information for all these widgets is printed as well.

Returns `out` : {None or dict}

In its default parameterization this method returns None. Depending on the parameter dictionary it is also possible that this method returns a dictionary with the single components of the information text.

`itom.filterHelp([filterName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given filter(s).

This method prints information about one specific filter (algorithm) or a list of filters to the console output. If one specific filter, defined in an algorithm plugin can be found that case-sensitively fits the given filterName its full documentation is printed. Else, a list of filters is printed whose name contains the given filterName.

Parameters `filterName` : {str}, optional

is the fullname or a part of any filter-name which should be displayed. If filterName is empty or no filter matches filterName (case sensitive) a list with all suitable filters is given.

`dictionary` : {dict}, optional

if dictionary == 1, a dictionary with all relevant components of the filter's documentation is returned and nothing is printed to the command line [default: 0]

`furtherInfos` : {int}, optional

Usually, filters or algorithms whose name only contains the given filterName are only listed at the end of the information text. If this parameter is set to 1 [default: 0], the full information for all these filters is printed as well.

Returns `out` : {None or dict}

In its default parameterization this method returns None. Depending on the parameter dictionary it is also possible that this method returns a dictionary with the single components of the information text.

`itom.pluginLoaded(pluginName)` → check if a certain plugin could be successfully loaded.

Checks if a specified plugin is loaded and returns the result as a boolean expression.

Parameters `pluginName` : {str}

The name of a specified plugin as usually displayed in the plugin window.

Returns `result` : {bool}

True, if the plugin has been loaded and can be used, else False.

`itom.version([toggle-output[, include-plugins]])` → retrieve complete information about itom version numbers

Parameters `toggle-output` : {bool}, optional

default = false if true, output will be written to a dictionary else to console.

`dictionary` : {bool}, optional

default = false if true, add informations about plugIn versions.

Returns None (display outPut) or PyDictionary with version information. :

Notes

Retrieve complete version information of itom and if specified version information of loaded plugins and print it either to the console or to a PyDictionary.

10.1.4 Adding elements to the GUI

`itom.addButton(toolbarName, buttonName, code[, icon, argtuple])` → adds a button to a toolbar in the main window

This function adds a button to a toolbar in the main window. If the button is pressed the given code, function

or method is executed. If the toolbar specified by 'toolbarName' does not exist, it is created. The button will show the optional icon, or if not given or not loadable, 'buttonName' is displayed as text.

itom comes with basic icons addressable by './../iconname.png', e.g. './gui/icons/close.png'. These natively available icons are listed in the icon-browser in the menu 'edit >> iconbrowser' of any script window. Furthermore you can give a relative or absolute path to any allowed icon file (the preferred file format is png).

Parameters **toolbarName** : {str}

The name of the toolbar.

buttonName : {str}

The name and identifier of the button to create.

code : {str, method, function}

The code to be executed if the button is pressed.

icon : {str}, optional

The filename of an icon-file. This can also be relative to the application directory of 'itom'.

argtuple : {tuple}, optional

Arguments, which will be passed to the method (in order to avoid cyclic references try to only use basic element types).

Returns **handle** : {int}

handle to the newly created button (pass it to `removeButton` to delete exactly this button)

Raises **Runtime error** :

if the main window is not available

See also:

`removeButton`

`itom.removeButton(handle | toolbarName[, buttonName])` → removes a button from a given toolbar.

This method removes an existing button from a toolbar in the main window of 'itom'. This button must have been created using *`addButton`*. If the toolbar is empty after the removal, it is finally deleted.

Pass either the 'handle' parameter of both 'toolbarName' and 'buttonName'. It is more precise to use the handle in order to exactly delete the button that has been created by a call to *`addButton`*. Using the names of the toolbar and the button always delete any button that has been created using this data.

Parameters **handle** : {int}

The handle returned by `addButton()`.

toolbarName : {str}

The name of the toolbar.

buttonName : {str}

The name (str, identifier) of the button to remove.

Raises **Runtime error** :

if the main window is not available or the given button could not be found.

See also:

`addButton`

`itom.addMenu(type, key[, name, code, icon, argtuple])` → adds an element to the menu bar of itom.

This function adds an element to the main window menu bar. The root element of every menu-list must be a MENU-element. Such a MENU-element can contain sub-elements. The following sub-elements can be

either another MENU, a SEPARATOR or a BUTTON. Only the BUTTON itself triggers a signal, which then executes the code, given by a string or a reference to a callable python method or function. Remember, that this reference is only stored as a weak pointer. If you want to directly add a sub-element, you can give a slash-separated string in the key-parameter. Every sub-component of this string then represents the menu-element in its specific level. Only the element in the last can be something else than MENU.

itom comes with basic icons addressable by `'../iconname.png'`, e.g. `'../gui/icons/close.png'`. These natively available icons are listed in the icon-browser in the menu `'edit >> iconbrowser'` of any script window. Furthermore you can give a relative or absolute path to any allowed icon file (the preferred file format is png).

Parameters **type** : {Int}

The type of the menu-element (BUTTON:0 [default], SEPARATOR:1, MENU:2). Use the corresponding constants in module `'itom'`.

key : {str}

A slash-separated string where every sub-element is the key-name for the menu-element in the specific level.

name : {str}, optional

The text of the menu-element. If not indicated, the last sub-element of key is taken.

code : {str, Method, Function}, optional

The code to be executed if menu element is pressed.

icon : {str}, optional

The filename of an icon-file. This can also be relative to the application directory of `'itom'`.

argtuple : {tuple}, optional

Arguments, which will be passed to method (in order to avoid cyclic references try to only use basic element types).

Returns **handle** : {int}

Handle to the recently added leaf node (action, separator or menu item). Use this handle to delete the item including its child items (for type `'menu'`).

Raises **Runtime error** :

if the main window is not available or the given button could not be found.

See also:

[`removeMenu`](#)

`itom.removeMenu(key | menuHandle)` → remove a menu element with the given key or handle.

This function remove a menu element with the given key or menuHandle. key is a slash separated list. The sub-components then lead the way to the final element, which should be removed.

Alternatively, it is possible to pass the handle obtained by [`addMenu`](#).

Parameters **key** : {str}, optional

The name (str, identifier) of the menu entry to remove.

handle : {int}, optional

The handle of the menu entry that should be removed (including its possible child items).

Raises **Runtime error** :

if the main window is not available or the given button could not be found.

See also:

[`addMenu`](#)

For more information about using these methods, see *Customize the menu and toolbars of itom*.

10.1.5 Disk-IO

`itom.loadDataObject(filename, dataObject[, doNotAppendIDO])` → load a dataObject from the harddrive.

This function reads a *dataObject* from the file specified by filename. MetaData saveType (string, binary) are extracted from the file and restored within the object.

Parameters `filename` : {str}

Filename and Path of the destination (.ido will be added if not available)

dataObject : {*dataObject*}

A pre-allocated *dataObject* (empty dataObject is allowed).

doNotAppendIDO : {bool}, optional

False[default]: file suffix *.ido* will not be appended to filename, True: it will be added.

Notes

The value of string-Tags must be encoded to avoid XML-conflicts. Tagnames which contains special characters leads to XML-conflicts.

`itom.saveDataObject(filename, dataObject[, tagsAsBinary = False])` → save a dataObject to hard-drive in a xml-based file format.

This method writes a *dataObject* into the file specified by 'filename'. The data is stored in a binary format within a xml-based structure. All string-tags of the dataObject are encoded in order to avoid xml-errors, the value of numerical tags are converted to string with 15 significant digits (>32bit, tagsAsBinary = False [default]) or in a binary format (tagsAsBinary = True).

Parameters `filename` : {str}

Filename and Path of the destination (.ido will be added if no *.**-ending is available)

dataObject : {DataObject}

An allocated dataObject of n-Dimensions.

tagsAsBinary : {bool}, optional

Optional tag to toggle if numeric-tags should be saved (metaData) as binary or by default as string.

See also:

loadDataObject

Notes

Tagnames which contains special characters leads to XML-conflicts.

`itom.loadMatlabMat(filename)` → loads Matlab mat-file by using scipy methods and returns the loaded dictionary.

This function loads matlab mat-file by using scipy methods and returns the loaded dictionary.

Parameters `filename` : {str}

Filename from which the data will be imported (.mat will be added if not available)

Returns `mat` : {dict}

dictionary with content of file

See also:`saveMatlabMat`

itom.**saveMatlabMat** (*filename*, *values*[, *matrixName* = 'matrix']) → save strings, numbers, arrays or combinations into a Matlab mat file.

Save one or multiple objects (strings, numbers, arrays, `dataObject`, `numpy.ndarray`, `npDataObject`...) to a Matlab *mat* file. There are the following possibilities for saving:

- One given value is saved under one given 'matrixName' or 'matrix' if 'matrixName' is not given.
- A list or tuple of objects is given. If no 'matrixName' is given, the items get the names 'matrix1', 'matrix2' ... Else, 'matrixName' must be a sequence of value names with the same length than 'values'.
- A dictionary is given, such that each value is stored under its corresponding key.

Parameters filename : {str}

Filename under which the file should be saved (.mat will be appended if not available)

values : {dictionary, list, tuple, variant}

single value, dictionary, list or tuple with elements of type number, string, array (`dataObject`, `numpy.ndarray`, `npDataObject`...)

matrix-name : {str, list, tuple}, optional

if 'values' is a single value, this parameter must be one single str, if 'values' is a sequence it must be a sequence of strings with the same length, if 'values' is a dictionary this argument is ignored.

See also:`loadMatlabMat`

itom.**loadIDC** (*filename*) → load a pickled idc-file and return the content as dictionary

This methods loads the given idc-file using the method `pickle.load` from the python-buildin module `pickle` and returns the loaded dictionary.

Parameters filename : {String}

absolute filename or filename relative to the current directory.

Returns content : {dict}

dictionary with loaded content

See also:`pickle.load`, `saveIDC`

itom.**saveIDC** (*filename*, *dict*[, *overwriteIfExists* = True]) → saves the given dictionary as pickled idc-file.

This method saves the given dictionary as pickled idc-file using the method `dump` from the builtin module `pickle`.

Parameters filename : {string}

absolute filename or filename relative to the current directory.

dict : {dict}

dictionary which should be pickled.

overwriteIfExists : {bool}, default: True

if True, an existing file will be overwritten.

See also:`pickle.dump`, `loadIDC`

10.1.6 Debug-Tools

`itom.gcEndTracking()` → compares the current object list of the garbage collector with the recently saved list.

`itom.gcStartTracking()` → stores the current object list of the garbage collector.

`itom.getDebugger()` → returns new reference to debugger instance

10.1.7 Further commands

`itom.scriptEditor()` → opens new, empty script editor window (undocked)

`itom.openScript(filename)` → open the given script in current script window.

Open the python script indicated by *filename* in a new tab in the current, latest opened editor window. Filename can be either a string with a relative or absolute filename to the script to open or any object with a `__file__` attribute. This attribute is then read and used as path.

The relative filename is relative with respect to the current directory.

Parameters `filename` : {str} or {obj}

Relative or absolute filename to a python script that is then opened (in the current editor window). Alternatively an object with a `__file__` attribute is allowed.

`itom.newScript()` → opens an empty, new script in the current script window.

Creates a new itom script in the latest opened editor window.

`itom.setCurrentPath(newPath)` → set current working directory to given absolute newPath

sets the absolute path of the current working directory to 'newPath'. The current working directory is the base directory for all subsequent relative pathes of icon-files, script-files, ui-files, relative import statements...

The current directory is always indicated in the right corner of the status bar of the main window.

Parameters `newPath` : {str}

The new working path of this application

Returns `success` : {bool}

True in case of success else False

See also:

[`getCurrentPath`](#)

`itom.getAppPath()` → returns absolute path of application base directory.

This function returns the absolute path of application base directory. The return value is independent of the current working directory.

Returns `path` : {str}

absolute path of this application's base directory

`itom.getCurrentPath()` → returns absolute path of current working directory.

Returns `Path` : {str}

absolute path of current working directory

See also:

[`setCurrentPath`](#)

`itom.getScreenInfo()` → returns dictionary with information about all available screens.

This method returns a dictionary with information about the current screen configuration of this computer.

Returns `screenInfo` : {dict}

dictionary with the following content is returned:

- `screenCount` (int): number of available screens
- `primaryScreen` (int): index (0-based) of primary screen
- `geometry` (tuple): tuple with dictionaries for each screen containing data for width (w), height (h) and its top-left-position (x, y)

`itom.getDefaultScaleableUnits()` → Get a list with the strings of the standard scalable units.

The unit strings returned as a list by this method can be transformed into each other using `scaleValueAndUnit`.

Returns `units` : {list}

List with strings containing all scaleable units

See also:

`scaleValueAndUnit`

`itom.scaleValueAndUnit()`

`ScaleValueAndUnit(scaleableUnits, value, valueUnit)` → Scale a value and its unit and returns [value, 'Unit']

Parameters `scaleableUnits` : {PyList of Strings}

A string list with all scaleable units

value : {double}

The value to be scaled

valueUnit : {str}

The value unit to be scaled

Returns `PyTuple with scaled value and scaled unit` :

Notes

Rescale a value with SI-unit (e.g. 0.01 mm to 10 micrometer). Used together with `itom.getDefaultScaleableUnits()`

`itom.clc()` → *clears the itom command line (if available)*

Since, the default-role property in `conf.py` is set to 'autolink' and the auto-summary module is included, small pages will be automatically created for each method in the following list and a hyperlink to this site is created:

10.2 dataIO

`class itom.dataIO(name[, mandparams, optparams])` → creates new instance of dataIO plugin

This is the constructor for a `dataIO` plugin. It initializes a new instance of the plugin specified by 'name'. The initialisation parameters are parsed and unnamed parameters are used in their incoming order to fill first mandatory parameters and afterwards optional parameters. Parameters may be passed with name as well but after the first named parameter no more unnamed parameters are allowed.

See `pluginHelp(name)` for detail information about the specific initialisation parameters.

Parameters `name` : {str}

is the fullname (case sensitive) of an 'actuator'-plugin as specified in the plugin-window.

mandparams : {variant(s)}

arguments corresponding the mandatory initialization parameters. The number of arguments and their order must fit the the required mandatory parameters

optparams : {variant(s)}, optional

argument corresponding to the optional initialization parameters. If unnamed arguments are used, their order must correspond to the order of the optional parameters, keyword-based parameters are allowed as well.

Returns **inst** : {dataIO}

new instance of the dataIO-plugin

acquire (*trigger=dataIO.TRIGGER_SOFTWARE*) → triggers a new the camera acquisition

This method triggers a new data acquisition. This method immediately returns even if the acquisition is not finished yet. Use *getVal* or *copyVal* to get the acquired data. Both methods block until the data is available.

Parameters **trigger** : {int}, optional

Type of the trigger:

- *dataIO.TRIGGER_SOFTWARE* = 0 : a software trigger is started, hence, the acquisition is immediately started when calling this method
- others : depending on your camera, this parameter can be used to set other triggers, like hardware trigger with raising or falling edges...

copyVal (*dataObject*) → gets deep copy of data of this plugin, stored in the given data object.

Returns a deep copy of the recently acquired data (for grabber and ADDA only) of the camera or AD-converter device. The deep copy sometimes requires one copy operation more than the similar command *getVal*. However, *getVal* only returns a reference to the plugin internal data structure whose values might be changed if another data acquisition is started.

If no acquisition has been triggered, this method raises a `RuntimeError`. If the acquisition is not finished yet, this method blocks and waits until the end of the acquisition.

Parameters **dataObject** : {*dataObject*}

dataObject where the plugin data is copied to. Either provide an empty *dataObject* or a *dataObject* whose size (or region of interest) exactly has the same size than the available data of the plugin. Therefore you can allocate a 3D data object, set a region of interest to one plane such that the data from the plugin is copied into this plane.

Raises **RuntimeError** :

if the dataIO plugin is anything else than ADDA or grabber or if no acquisition has been triggered

See also:

getVal

disableAutoGrabbing () → Disable auto grabbing for the grabber (camera...),

If the auto grabbing flag is set, the camera acquisition is automatically and continuously triggered if at least one live image is connected. This is an undesired behaviour if a measurement is started where the acquisition should be controlled by a specific script or something similar. Then disable the auto grabbing property. All connected live images will then get new images when *getVal* or *copyVal* is called by the script. The live image timer is disabled.

This method disables the auto grabbing flag.

See also:

setAutoGrabbing, *disableAutoGrabbing*, *getAutoGrabbing*

enableAutoGrabbing () → enable auto grabbing for the grabber (camera...),

If the auto grabbing flag is set, the camera acquisition is automatically and continuously triggered if

at least one live image is connected. This is an undesired behaviour if a measurement is started where the acquisition should be controlled by a specific script or something similar. Then disable the auto grabbing property. All connected live images will then get new images when `getVal` or `copyVal` is called by the script. The live image timer is disabled.

This method enables the auto grabbing flag.

See also:

`setAutoGrabbing`, `disableAutoGrabbing`, `getAutoGrabbing`

exec (`funcName` [, `param1`, ...]) → invoke the function ‘funcName’ registered as `execFunc` within the plugin.

Every plugin can define further functions that can for instance be used in order to call specific calibration routines of cameras or actuators. This general method is used to call one of these specific functions registered under `funcName`.

Parameters `funcName` : {str}

The name of the function

param1 : {variant}, optional

Further parameters depending on the requirements of the specific function.

Returns `out` : {variant, list of variants}.

The return values depend on the function itself.

See also:

`execFuncsInfo`

getAutoGrabbing () → return the status of the auto grabbing flag.

If the auto grabbing flag is set, the camera acquisition is automatically and continuously triggered if at least one live image is connected. This is an undesired behaviour if a measurement is started where the acquisition should be controlled by a specific script or something similar. Then disable the auto grabbing property. All connected live images will then get new images when `getVal` or `copyVal` is called by the script. The live image timer is disabled.

Returns `auto grabbing flag` : {bool}

- False = auto grabbing off
- True = auto grabbing on.

See also:

`enableAutoGrabbing`, `disableAutoGrabbing`, `setAutoGrabbing`

getExecFuncsInfo ([`funcName` [, `detailLevel`]]) → plots a list of available `execFuncs` or a detailed description of the specified `execFunc`.

Every plugin can define further functions, that are called by `plugin.exec(‘funcName’ [,param1, param2...])`. This can for instance be used in order to call specific calibration routines of cameras or actuators. This method allows printing information about available functions of this type.

Parameters `funcName` : {str}, optional

is the fullname or a part of any `execFunc`-name which should be displayed. If `funcName` is none or no `execFunc` matches `funcName` casesensitiv a list with all suitable `execFuncs` is given.

detailLevel : {dict}, optional

if `detailLevel == 1`, function returns a dictionary with parameters [default: 0].

Returns `out` : {None or dict}

depending on the value of `detailLevel`.

See also:*exec***getParam** (*name*) → current value of the plugin parameter 'name'.

Returns the current value of the internal plugin parameter with 'name'. The type of the returned value depends on the real type of the internal plugin, which may be:

- String -> str
- Char, Integer -> int
- Double -> float
- CharArray, IntegerArray -> tuple of int
- DoubleArray -> tuple of float
- DataObject -> dataObject
- PolygonMesh -> polygonMesh
- PointCloud -> pointCloud
- Another plugin instance -> dataIO or actuator

The name of the parameter must have the following form:

- name
- name:additionalTag (additionalTag can be a special feature of some plugins)
- name[index] (only possible if parameter is an array type and you only want to get one single value, specified by the integer index [0,nrOfArrayItems-1])
- name[index]:additionalTag

Parameters *name* : {str}

name of the requested parameter

Returns *out* : {variant}

value of the parameter

Raises **ValueError** :

if parameter does not exist

See also:*setParam, getParamList, getParamListInfo***getParamInfo** (*name*) → returns dictionary with meta information of parameter 'name'.**getParamList** () → returns a list of the names of the internal parameters of the plugin

Each plugin defines a set of parameters, where each parameter has got a name and maps to any value.

The value is represented by the C++ class `ito::ParamBase` and can have one of the following types:

- String
- Char
- Integer
- Double
- CharArray
- IntegerArray
- DoubleArray
- DataObject

- PolygonMesh
- PointCloud
- Another plugin instance

Using one of the parameter names, its current value can be obtained by `getParam('name')` and is writable by `setParam('name', newValue)` (if not read-only)

Returns out : {list}

list of parameter names

See also:

`getParam`, `setParam`, `getParamListInfo`

getParamListInfo (`[detailLevel]`) → prints detailed information about all plugin parameters.

Each plugin defines a set of parameters, where each parameter has got a name and maps to any value. The value is represented by the C++ class `ito::ParamBase` and can have one of the following types:

- String
- Char
- Integer
- Double
- CharArray
- IntegerArray
- DoubleArray
- DataObject
- PolygonMesh
- PointCloud
- Another plugin instance

Using one of the parameter names, its current value can be obtained by `getParam('name')` and is writable by `setParam('name', newValue)` (if not read-only)

This method prints a detailed table with the name, current value, description string and further meta information of every plugin parameter. Additionally, the column R/W indicates if this parameter is writable or read-only.

Parameters detailLevel : {dict}, optional

if `detailLevel == 1`, function returns a dictionary with parameters, else None is returned and the output is printed in a readable form to the console [default]

Returns out : {None, dict}

If `detailLevel == 1`, a dictionary containing all printed information is returned

See also:

`getParam`, `setParam`, `getParamInfo`, `getParamList`

getType () → returns dataIO type

getVal (`buffer='dataObject'|'bytearray'|'bytes'[, length=maxlength]`) → returns shallow copy of internal camera image if 'dataObject'-buffer is provided. Else values from plugin are copied to given byte or byte-array buffer.

Returns a reference (shallow copy) of the recently acquired image (located in the internal memory if the plugin) if the plugin is a grabber or camera and the buffer is a `dataObject`. Please consider that the values of the `dataObject` might change if a new image is acquired since it is only a reference. Therefore consider copying the `dataObject` or directly use `copyVal`.

If no acquisition has been triggered, this method raises a `RuntimeError`. If the acquisition is not finished yet, this method blocks and waits until the end of the acquisition.

If the plugin is another type than a grabber or camera (e.g. `serialIO`), this method requires any buffer-object that is preallocated with a reasonable size. Then, the currently available data is copied into this buffer object and the size of the copied data is returned. If the buffer is too small, only the data that fits into the buffer is copied. Another call to `getVal` will copy the rest.

Parameters `buffer` : {`dataObject`, `bytearray`, `bytes` or `str`}

this parameter depends on the type of `dataIO`-instance:

- cameras, grabber: the buffer must be a `dataObject` (no length parameter): A reference (shallow copy) to the internal memory of the camera plugin is set to the given data object. Therefore its content may change if a new image is being acquired by the camera. Consider taking a deep copy if the image (`dataObject.copy`) or use the method `copyVal`.
- other IO-devices (AD-converters): The buffer must be an object of type `dataObject`, `bytearray`, `bytes` or unicode string. The length parameter is then set to the size of the buffers. The effective size of the used memory in buffer is returned.

length : {int}, optional

size of the given buffer. This value is usually automatically determined and must not be given.

Returns `out` : {None or int}

None or size of used buffer if buffer is no `dataObject`

See also:

`copyVal`

hideToolbox () → hides toolbox of the plugin

Raises `RuntimeError` :

if plugin does not provide a toolbox

See also:

`showToolbox`

name () → returns the plugin name

Returns `name` : {str}

name of the plugin, which corresponds to `getParam('name')`

See also:

`getParam`

setAutoGrabbing (`on`) → Set auto grabbing of the grabber device to on or off

If the auto grabbing flag is set, the camera acquisition is automatically and continuously triggered if at least one live image is connected. This is an undesired behaviour if a measurement is started where the acquisition should be controlled by a specific script or something similar. Then disable the auto grabbing property. All connected live images will then get new images when `getVal` or `copyVal` is called by the script. The live image timer is disabled.

This method allows setting this flag.

Parameters `on` : {bool}

- `TRUE` = on
- `FALSE` = off

See also:

enableAutoGrabbing, disableAutoGrabbing, getAutoGrabbing

setParameter (*name, value*) → sets parameter ‘name’ to the given value.

Sets the internal plugin parameter with ‘name’ to a new value. The plugin itself can decide whether the given value is accepted as new value. This may depend on the type of the given value, but also on the allowed value range indicated by further meta information of the internal parameter. Parameters that have the read-only flag set can not be reset.

The name of the parameter must have the following form:

- name
- name:additionalTag (additionalTag can be a special feature of some plugins)
- name[index] (only possible if parameter is an array type and you only want to get one single value, specified by the integer index [0,nrOfArrayItems-1])
- name[index]:additionalTag

Parameters **name** : {str}

name of the parameter

value : {str, int, double, ...}

value that will be set. Only the name and existence of the parameter is checked before passing the request to the plugin. The plugin itself is responsible for further validations (including read-only attribute).

See also:

getParam, getParamList, getParamListInfo

setVal (*dataObjectOrBuffer* [, *length=1*]) → transfer given ‘dataObject’ to ADDA plugin or further buffer to other dataIO plugin.

If the dataIO plugin has the subtype ADDA, this method is used to send data to one or more analog outputs of the device. In this case a *dataObject* must be given as first argument and the second argument *length* must be 1.

For other dataIO plugins, the first argument must be any buffer object, like *bytearray*, *bytes* or *unicode string*. The length is then extracted from this value. However it is also possible to define a user-defined size using the ‘length’ argument.

Parameters **dataObjectOrBuffer** : {*dataObject*, *bytearray*, *bytes*, *str*}

value to send to plugin. For an ADDA plugin, a *dataObject* is required whose content is sent to the analogous outputs of the device. For other dataIO plugins buffer values like *bytearray*, *bytes* or *unicode string* are required.

length : {int}, optional

usually this value is not required, since the length of the buffer is automatically extracted from the given objects and 1 for a *dataObject*

showConfiguration () → show configuration dialog of the plugin

Raises **RuntimeError** :

if plugin does not provide a configuration dialog

showToolbox () → open toolbox of the plugin

Raises **RuntimeError** :

if plugin does not provide a toolbox

See also:

hideToolbox

startDevice ($[count=1]$) \rightarrow starts the given dataIO-plugin.

This command starts the dataIO plugin such that it is ready for data acquisition. Call this method before you start using commands like `acquire`, `getVal` or `copyVal`. If the device already is started, an internal start-counter is incremented by the parameter ‘count’. The corresponding `stopDevice` method then decrements this counter and finally stops the device once the counter drops to zero again.

The counter is necessary, since every connected live image needs to start the device without knowledge about any previous start. No acquisition is possible, if the device has not been started, hence the counter is 0.

Parameters count : {int}, optional

Number of increments to the internal start-counter [default:1]

See also:

stopDevice

stopDevice ($[count=1]$) \rightarrow stops the given dataIO-plugin.

If this method is called as many times as the corresponding `startDevice` (or if the counts are equal), the dataIO device is stopped (not deleted) and it is not possible to acquire further data.

Once a live image is connected to a camera, `startDevice` is automatically called at start of the live acquisition and `stopDevice` at shutdown.

Parameters count : {int}, optional

default = 1 if count > 1, `stopDevice` is executed ‘count’ times, in order to decrement the grabber internal start counter. You can also use -1 as count argument, then `stopDevice` is repeated until the internal start counter is 0. The number of effective counts is then returned

Returns counts : {None or int}

If *count* == -1 the number of required counts to finally stop the device is returned. Else:
None

See also:

startDevice

10.3 actuator

class `itom.actuator` (*name*[, *mandparams*, *optparams*]) → creates new instance of actuator plugin
 ‘name’

This is the constructor for an *actuator* plugin. It initializes a new instance of the plugin specified by 'name'. The initialisation parameters are parsed and unnamed parameters are used in their incoming order to fill first mandatory parameters and afterwards optional parameters. Parameters may be passed with name as well but after the first named parameter no more unnamed parameters are allowed.

See `pluginHelp(name)` for detail information about the specific initialisation parameters.

Parameters **name** : {str}

is the fullname (case sensitive) of an ‘actuator’-plugin as specified in the plugin-window.

mandparams : { variant(s) }

arguments corresponding the mandatory initialization parameters. The number of arguments and their order must fit the the required mandatory parameters

optparams : { variant(s) }, optional

argument corresponding to the optional initialization parameters. If unnamed arguments are used, their order must correspond to the order of the optional parameters, keyword-based parameters are allowed as well.

Returns `inst` : {actuator}

new instance of the actuator-plugin

calib (`axis`[, `axis1`, ...]) → starts calibration or homing of given axes (0-based).

Most actuators have the possibility to calibrate or home certain axes. Use this command to start the calibration.

Parameters `axis` : {int}

index of the first axis to calibrate (e.g. 0 for first axis)

`axis1` : {int}

add the indices of further axes as optional arguments if they should be calibrated as well

Raises `NotImplemented` :

if calibration not available

exec (`funcName`[, `param1`, ...]) → invoke the function ‘funcName’ registered as `execFunc` within the plugin.

Every plugin can define further functions that can for instance be used in order to call specific calibration routines of cameras or actuators. This general method is used to call one of these specific functions registered under `funcName`.

Parameters `funcName` : {str}

The name of the function

`param1` : {variant}, optional

Further parameters depending on the requirements of the specific function.

Returns `out` : {variant, list of variants}.

The return values depend on the function itself.

See also:

`execFuncsInfo`

getExecFuncsInfo ([`funcName`[, `detailLevel`]]) → plots a list of available `execFuncs` or a detailed description of the specified `execFunc`.

Every plugin can define further functions, that are called by `plugin.exec(‘funcName’ [,param1, param2...])`. This can for instance be used in order to call specific calibration routines of cameras or actuators. This method allows printing information about available functions of this type.

Parameters `funcName` : {str}, optional

is the fullname or a part of any `execFunc`-name which should be displayed. If `funcName` is none or no `execFunc` matches `funcName` casesensitiv a list with all suitable `execFuncs` is given.

`detailLevel` : {dict}, optional

if `detailLevel == 1`, function returns a dictionary with parameters [default: 0].

Returns `out` : {None or dict}

depending on the value of `detailLevel`.

See also:

`exec`

getParam (`name`) → current value of the plugin parameter ‘name’.

Returns the current value of the internal plugin parameter with ‘name’. The type of the returned value depends on the real type of the internal plugin, which may be:

- String -> str
- Char, Integer -> int

- Double -> float
- CharArray, IntegerArray -> tuple of int
- DoubleArray -> tuple of float
- DataObject -> dataObject
- PolygonMesh -> polygonMesh
- PointCloud -> pointCloud
- Another plugin instance -> dataIO or actuator

The name of the parameter must have the following form:

- name
- name:additionalTag (additionalTag can be a special feature of some plugins)
- name[index] (only possible if parameter is an array type and you only want to get one single value, specified by the integer index [0,nrOfArrayItems-1])
- name[index]:additionalTag

Parameters **name** : {str}

name of the requested parameter

Returns **out** : {variant}

value of the parameter

Raises **ValueError** :

if parameter does not exist

See also:

setParam, getParamList, getParamListInfo

getParamInfo (*name*) → returns dictionary with meta information of parameter 'name'.

getParamList () → returns a list of the names of the internal parameters of the plugin

Each plugin defines a set of parameters, where each parameter has got a name and maps to any value.

The value is represented by the C++ class `ito::ParamBase` and can have one of the following types:

- String
- Char
- Integer
- Double
- CharArray
- IntegerArray
- DoubleArray
- DataObject
- PolygonMesh
- PointCloud
- Another plugin instance

Using one of the parameter names, its current value can be obtained by *getParam('name')* and is writable by *setParam('name', newValue)* (if not read-only)

Returns **out** : {list}

list of parameter names

See also:*getParam, setParam, getParamListInfo***getParamListInfo** (*[detailLevel]*) → prints detailed information about all plugin parameters.

Each plugin defines a set of parameters, where each parameter has got a name and maps to any value. The value is represented by the C++ class `ito::ParamBase` and can have one of the following types:

- String
- Char
- Integer
- Double
- CharArray
- IntegerArray
- DoubleArray
- DataObject
- PolygonMesh
- PointCloud
- Another plugin instance

Using one of the parameter names, its current value can be obtained by *getParam('name')* and is writable by *setParam('name', newValue)* (if not read-only)

This method prints a detailed table with the name, current value, description string and further meta information of every plugin parameter. Additionally, the column R/W indicates if this parameter is writable or read-only.

Parameters *detailLevel* : {dict}, optional

if *detailLevel == 1*, function returns a dictionary with parameters, else None is returned and the output is printed in a readable form to the console [default]

Returns *out* : {None, dict}

If *detailLevel == 1*, a dictionary containing all printed information is returned

See also:*getParam, setParam, getParamInfo, getParamList***getPos** (*axis* [*axis1*, ...]) → returns the actual positions of the given axes (in mm or degree).

This method requests the current position(s) of the given axes and returns it.

Parameters *axis* : {int}

index of the first axis (e.g. 0 for first axis)

axis1 : {int}

add the indices of further axes as optional arguments

Returns *positions* : {float or tuple of float}

Current position as float value if only one axis is given or tuple of floats for multiple axes. The unit is mm or degree.

See also:*setPosRel, setPosAbs***getStatus** () → returns a list of status values for each axis

Each axis of an actuator plugin has got a status value that is used for informing about the current status of the axis.

The status value is an or-combination of the following possible values:

Moving flags:

- `actuatorUnknown = 0x0001` : unknown current moving status
- `actuatorInterrupted = 0x0002` : movement has been interrupted by the user or another error during the movement occurred
- `actuatorMoving = 0x0004` : axis is currently moving
- `actuatorAtTarget = 0x0008` : axis reached the target position
- `actuatorTimeout = 0x0010` : timeout during movement. Unknown status of the movement

Switches flags:

- `actuatorEndSwitch = 0x0100` : axis reached any end switch (e.g. if only one end switch is available)
- `actuatorLeftEndSwitch = 0x0200` : axis reached the left end switch
- `actuatorRightEndSwitch = 0x0400` : axis reached the right end switch
- `actuatorRefSwitch = 0x0800` : axis reached any reference switch (e.g. for calibration...)
- `actuatorLeftRefSwitch = 0x1000` : axis reached left reference switch
- `actuatorRightRefSwitch = 0x2000` : axis reached right reference switch

Status flags:

- `actuatorAvailable = 0x4000` : the axis is available
- `actuatorEnabled = 0x8000` : the axis is currently enabled and can be moved

Returns `status` : {list of integers}

list of integers (size corresponds to number of axes) with the current status of each axis

`getType()` → returns actuator type

`hideToolbox()` → hides toolbox of the plugin

Raises `RuntimeError` :

if plugin does not provide a toolbox

See also:

[`showToolbox`](#)

`name()` → returns the plugin name

Returns `name` : {str}

name of the plugin, which corresponds to `getParam('name')`

See also:

[`getParam`](#)

`setInterrupt()` → interrupts a movement of an actuator

Sets the interrupt flag of an actuator. The actuator interrupts the movement of all running axes as soon as this flag is checked again.

`setOrigin(axis[, axis1, ...])` → defines the actual position of the given axes to value 0.

The current positions of all indicated axes (axis, axis1,...) are considered to be 0 such that following positioning commands are relative with respect to the current position.

Parameters `axis` : {int}

index of the first axis (e.g. 0 for first axis)

axis1 : {int}

add the indices of further axes as optional arguments

Raises NotImplemented :

if actuator does not support this feature

setParam (*name*, *value*) → sets parameter ‘name’ to the given value.

Sets the internal plugin parameter with ‘name’ to a new value. The plugin itself can decide whether the given value is accepted as new value. This may depend on the type of the given value, but also on the allowed value range indicated by further meta information of the internal parameter. Parameters that have the read-only flag set can not be reset.

The name of the parameter must have the following form:

- name
- name:additionalTag (additionalTag can be a special feature of some plugins)
- name[index] (only possible if parameter is an array type and you only want to get one single value, specified by the integer index [0,nrOfArrayItems-1])
- name[index]:additionalTag

Parameters name : {str}

name of the parameter

value : {str, int, double, ...}

value that will be set. Only the name and existence of the parameter is checked before passing the request to the plugin. The plugin itself is responsible for further validations (including read-only attribute).

See also:

[*getParam*](#), [*getParamList*](#), [*getParamListInfo*](#)

setPosAbs (*axis0*, *pos0*[, *axis1*, *pos1*, ...]) → moves given axes to given absolute values (in mm or degree).

All arguments are a pair of axis index and the new target position of this axis. This method starts the absolute positioning of all given axes. If the ‘async’ parameter of the plugin is 0 (usually default), a synchronous positioning is started, hence, this method returns after that all axes reached their target position or a timeout occurred. Else this method immediately returns and the actuator goes on moving.

Parameters axisM : {int}

index of the axis to position

posM : {float}

absolute target position of the *axisM* (in mm or degree)

See also:

[*getPos*](#), [*setPosRel*](#)

setPosRel (*axis0*, *pos0*[, *axis1*, *pos1*, ...]) → relatively moves given axes by the given distances [in mm or degree].

All arguments are a pair of axis index and the relative moving-distance of this axis. This method starts the relative positioning of all given axes. If the ‘async’ parameter of the plugin is 0 (usually default), a synchronous positioning is started, hence, this method returns after that all axes reached their target position or a timeout occurred. Else this method immediately returns and the actuator goes on moving.

Parameters axisM : {int}

index of the axis to position

posM : {float}

relative target position of the *axisM* (in mm or degree)

See also:

getPos, setPosAbs

showConfiguration () → show configuration dialog of the plugin

Raises RuntimeError :

if plugin does not provide a configuration dialog

showToolbox () → open toolbox of the plugin

Raises RuntimeError :

if plugin does not provide a toolbox

See also:

hideToolbox

10.4 Algorithms, Widgets and Filters

The algorithm-plugins are containers for **filters** and **widgets**. They do not have an own instance.

Filters are called by the **filter(...)**-Methods. To get an online help use **filterHelp(...)**. Widgets are used with **ui**-dialogs. To get an online help use **widgetHelp(...)**.

`itom.filter(name[, furtherParameters, ...])` → invoke a filter (or algorithm) function from an algorithm-plugin.

This function is used to invoke itom filter-functions or algorithms, declared within itom-algorithm plugins. The parameters (arguments) depends on the specific filter function (see `filterHelp(name)`), By `filterHelp()` a list of available filter functions is retrieved.

Parameters name : {str}

The name of the filter

furtherParameters : {variant}

Further parameters depend on the filter-methods itself (give the mandatory and then optional parameters in their defined order).

Returns out : {variant}

The returned values depend on the definition of each filter. In general it is a tuple of all output parameters that are defined by the filter function.

See also:

filterHelp

`itom.filterHelp([filterName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given filter(s).

This method prints information about one specific filter (algorithm) or a list of filters to the console output. If one specific filter, defined in an algorithm plugin can be found that case-sensitively fits the given `filterName` its full documentation is printed. Else, a list of filters is printed whose name contains the given `filterName`.

Parameters filterName : {str}, optional

is the fullname or a part of any filter-name which should be displayed. If `filterName` is empty or no filter matches `filterName` (case sensitive) a list with all suitable filters is given.

dictionary : {dict}, optional

if `dictionary == 1`, a dictionary with all relevant components of the filter's documentation is returned and nothing is printed to the command line [default: 0]

furtherInfos : {int}, optional

Usually, filters or algorithms whose name only contains the given `filterName` are only listed at the end of the information text. If this parameter is set to 1 [default: 0], the full information for all these filters is printed as well.

Returns out : {None or dict}

In its default parameterization this method returns None. Depending on the parameter dictionary it is also possible that this method returns a dictionary with the single components of the information text.

`itom.widgetHelp ([widgetName, dictionary = 0, furtherInfos = 0])` → generates an online help for the given widget(s).

This method prints information about one specific widget or a list of widgets to the console output. If one specific widget, defined in an algorithm plugin can be found that case-sensitively fits the given widget-Name its full documentation is printed. Else, a list of widgets is printed whose name contains the given widgetName.

Parameters widgetName : {str}, optional

is the fullname or a part of any widget-name which should be displayed. If widgetName is empty or no widget matches widgetName (case sensitive) a list with all suitable widgets is given.

dictionary : {dict}, optional

if dictionary == 1, a dictionary with all relevant components of the widget's documentation is returned and nothing is printed to the command line [default: 0]

furtherInfos : {int}, optional

Usually, widgets whose name only contains the given widgetName are only listed at the end of the information text. If this parameter is set to 1 [default: 0], the full information for all these widgets is printed as well.

Returns out : {None or dict}

In its default parameterization this method returns None. Depending on the parameter dictionary it is also possible that this method returns a dictionary with the single components of the information text.

10.5 figure

class `itom.figure ([handle[, rows = 1, cols = 1]])` → creates figure window.

Bases: `itom.uiItem`

The class `itom.figure` represents a standalone figure window, that can have various subplots. If an instance of this class is created without further parameters a new figure is created and opened having one subplot area (currently empty) and the numeric handle to this figure is returned:

```
h = figure()
```

Subplots are arranged in a regular grid whose size is defined by the optional parameters 'rows' and 'cols'. If you create a figure instance with a given handle, the instance is either a reference to an existing figure that has got this handle or if it does not exist, a new figure with the desired handle is opened and the handle is returned, too.

Parameters handle : {int}

numeric handle of the desired figure.

rows : {int, default: 1}

number of rows this figure should have (defines the size of the subplot-grid)

cols : {int, default: 1}

number of columns this figure should have (defines the size of the subplot-grid)

static close (*handle* | 'all') -> *static method to close any specific or all open figures (unless any figure-instance still keeps track of them)*

This method closes and deletes any specific figure (given by handle) or all opened figures.

Parameters **handle** : {dataIO-Instance}

any figure handle (>0) or 'all' in order to close all opened figures

Notes

If any instance of class 'figure' still keeps a reference to any figure, it is only closed and deleted if the last instance is deleted, too.

hide () → hides figure without deleting it

liveImage (*cam* [, *areaIndex*, *className*, *properties*]) → shows a camera live image in the current or given area of this figure

Creates a plot-image (2D) and automatically grabs images into this window. This function is not blocking.

Parameters **cam** : {dataIO-Instance}

Camera grabber device from which images are acquired.

areaIndex: {int}, optional :

Area number where the plot should be put if subplots have been created

className : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

properties : {dict}, optional

optional dictionary of properties that will be directly applied to the plot widget.

plot (*data* [, *areaIndex*, *className*, *properties*]) → plots a dataObject in the current or given area of this figure

Plot an existing dataObject in not dockable, not blocking window. The style of the plot will depend on the object dimensions. If x-dim or y-dim are equal to 1, plot will be a lineplot else a 2D-plot.

Parameters **data** : {DataObject}

Is the data object whose region of interest will be plotted.

areaIndex: {int}, optional :

Area number where the plot should be put if subplots have been created

className : {str}, optional

class name of desired plot (if not indicated default plot will be used (see application settings))

properties : {dict}, optional

optional dictionary of properties that will be directly applied to the plot widget.

show () → shows figure

subplot (*index*) → returns plotItem of desired subplot

This method closes and deletes any specific figure (given by handle) or all opened figures.

Parameters **index** : {unsigned int}

index to desired subplot. The subplot at the top, left position has the index 0 whereas the index is incremented row-wise.

docked

dock status of figure (True/False)

this attribute controls the dock appearance of this figure. If it is docked, the figure is integrated into the main window of itom, else it is a independent window.

handle

returns handle of figure

10.6 plotItem

class `itom.plotItem` (*figure* | *uiItem* [, *subplotIdx*]) → instance of the plot or subplot of a figure.

Bases: `itom.uiItem`

Use can use this constructor to access any plot or subplot (if more than one plot) of a figure. The subplotIndex row-wisely addresses the subplots, beginning with 0.

As second possibility, the constructor can be used to cast 'uiItem' to 'plotItem' in order to access methods like 'pickPoints' or 'drawAndPickElement'.

Parameters *figure* : {??}

subplotIdx : {??} :

drawAndPickElements (*elementType*, *elementData*, [, *maxNrElements*]) → *method to let the user draw geometric elements on a plot (only if plot supports this)*

This method lets the user select one or multiple elements of type (up to *maxNrElements*) at the current plot (if the plot supports this).

Parameters *elementType* : {int}

The element type to plot according to `ito::PrimitiveContainer::tPrimitive`.

points : {DataObject}

resulting data object containing the 2D positions of the selected points [2 x *nrOfSelectedPoints*].

maxNrElements: {int}, optional :

let the user select up to this number of points [default: infinity]. Selection can be stopped pressing Space or Esc.

pickPoints (*points* [, *maxNrPoints*]) → *method to let the user pick points on a plot (only if plot supports this)*

This method lets the user select one or multiple points (up to *maxNrPoints*) at the current plot (if the plot supports this).

Parameters *points* : {DataObject}

resulting data object containing the 2D positions of the selected points [2 x *nrOfSelectedPoints*].

maxNrPoints: {int}, optional :

let the user select up to this number of points [default: infinity]. Selection can be stopped pressing Space or Esc.

10.7 ui-elements (ui, uiItem)

10.7.1 uiItem-Class

class `itom.uiItem` (...) → base class representing any widget of a graphical user interface

This class represents any widget (graphical, interactive element like a button or checkbox) on a graphical

user interface. An instance of this class provides many functionalities given by the underlying Qt system. For instance, it is possible to call a public slot of the corresponding widget, connect signals to specific python methods or functions or change properties of the widget represented by the instance.

The overall dialog or window as main element of a graphical user interface itself are instances of the class *ui*. However, they are derived from *uiItem*, since dialogs or windows internally are widgets as well.

Widgets placed at a user interface using the Qt Designer can be referenced by an *uiItem* instance by their specific `objectName`, assigned in the Qt Designer as well. As an example, a simple dialog with one button is created and the text of the button (`objectName: btn`) is set to OK:

```
dialog = ui('filename.ui', type=ui.TYPEDIALOG)
button = dialog.btn #here the reference to the button is obtained
button["text"] = "OK" #set the property text of the button
```

Information about available properties, signals and slots can be obtained using the method `info()` of *uiItem*.

Notes

It is not intended to directly instantiate this class. Either create a user interface using the class *ui* or obtain a reference to an existing widget (this is then an instance of *uiItem*) using the dot-operator of a parent widget or the entire user interface.

call (*slotOrPublicMethod*[, *argument1*, *argument2*, ...]) → calls any public slot of this widget or any accessible public method.

This method invokes (calls) a method of the underlying widget that is marked as public slot. Besides slots there are some public methods of specific widget classes that are wrapped by itom and therefore are callable by this method, too.

If only method is available, all arguments are tried to be cast to the requested types and the slot is called on conversion success. If the method has multiple overloaded possibilities in the underlying C++ classes, at first, it is intended to find the variant where all arguments can be strictly casted from Python types to the necessary C-types. If this fails, the next variant with a non-strict conversion is chosen.

Parameters *slotOrPublicMethod* : {str}

name of the slot or method

arguments : {various types}, optional

Here you must indicate every argument, that the definition of the slot indicates. The type must be convertible into the requested C++ based argument type.

See also:

info

Notes

If you want to know all possible slots of a specific widget, see the Qt help or call the member `info()` of the widget.

children ([*recursive* = *False*]) → returns dict with widget-based child items of this *uiItem*.

Each key -> value pair is object-name -> class-name). Objects with no object-name are omitted.

Parameters *recursive* : {bool}

True: all objects including sub-widgets of widgets are returned, False: only children of this *uiItem* are returned (default)

connect (*signalSignature*, *callableMethod*) → connects the signal of the widget with the given callable python method

This instance of *uiItem* wraps a widget, that is defined by a C++-class, that is finally derived from

QWidget. See Qt-help for more information about the capabilities of every specific widget. Every widget can send various signals. Use this method to connect any signal to any callable python method (bounded or unbounded). This method must have the same number of arguments than the signal and the types of the signal definition must be convertible into a python object.

Parameters **signalSignature** : {str}

This must be the valid signature, known from the Qt-method *connect* (e.g. 'clicked(bool)')

callableMethod : {python method or function}

valid method or function that is called if the signal is emitted.

See also:

disconnect, *invokeKeyboardInterrupt*

disconnect (*signalSignature*, *callableMethod*) → disconnects a connection which must have been established with exactly the same parameters.

Parameters **signalSignature** : {str}

callableMethod : {python method or function}

Notes

doctodo

exists () → returns true if widget still exists, else false.

getAttribute (*attributeNumber*) → returns specified attribute of corresponding widget.

Widgets have specific attributes that influence their behaviour. These attributes are contained in the Qt-enumeration Qt::WidgetAttribute. Use this method to query the current status of one specific attributes.

Important attributes are:

- Qt::WA_DeleteOnClose (55) -> deletes the widget when it is closed, else it is only hidden [default]
- Qt::WA_MouseTracking (2) -> indicates that the widget has mouse tracking enabled

Parameters **attributeNumber** : {int}

Number of the attribute of the widget to query (enum Qt::WidgetAttribute)

Returns **out** : {bool}

True if attribute is set, else False

See also:

setAttribute

getProperty (*propertyName* | *listOfPropertyNames*) -> returns tuple of requested properties (single property or tuple of properties)

Use this method or the operator [] in order to get the value of one specific property of this widget or of multiple properties. Multiple properties are given by a tuple or list of property names. For one single property, its value is returned as it is. If the property names are passed as sequence, a sequence of same size is returned with the corresponding values.

Parameters **property** : {string, string-list}

Name of one property or sequence (tuple,list...) of property names

Returns **out** : {variant, sequence of variants}

the value of one single property of a list of values, if a sequence of names is given as parameter.

See also:

setProperty

getPropertyInfo (*[propertyName]*) → returns information about the property ‘propertyName’ of this widget or all properties, if no name indicated.

Parameters *propertyName* : {tuple}, optional

getWindowFlags (*flags*) → gets window flags of corresponding widget.

The flags-value is an or-combination of the enumeration Qt::WindowType. See Qt documentation for more information.

Returns *flags {int}*: :

or-combination of Qt::WindowType describing the type and further hints of the user interface

See also:

setWindowFlags

info (*[verbose = 0]*) → prints information about properties, public accessible slots and signals of the wrapped widget.

Parameters *verbose* : {int}

0: only properties, slots and signals that do not come from Qt-classes are printed (default)
1: properties, slots and signals are printed up to Qt GUI base classes
2: all properties, slots and signals are printed

invokeKeyboardInterrupt (*signalSignature*) → connects the given signal with a slot immediately invoking a python interrupt signal.

Parameters *signalSignature* : {str}

This must be the valid signature, known from the Qt-method *connect* (e.g. ‘clicked(bool)’)

See also:

connect

Notes

If you use the connect method to link a signal with a python method or function, this method can only be executed if python is in an idle status. However, if you want raise the python interrupt signal if a specific signal is emitted, this interruption should be immediately invoked. Therefore

setAttribute (*attributeNumber, value*) → sets attribute of corresponding widget.

Widgets have specific attributes that influence their behaviour. These attributes are contained in the Qt-enumeration Qt::WidgetAttribute. Use this method to enable/disable one specific attribute.

Important attributes are:

- Qt::WA_DeleteOnClose (55) -> deletes the widget when it is closed, else it is only hidden [default]
- Qt::WA_MouseTracking (2) -> indicates that the widget has mouse tracking enabled

Parameters *attributeNumber* : {int}

Number of the attribute of the widget to set (enum Qt::WidgetAttribute)

value : {bool}

True if attribute should be enabled, else False

See also:

getAttribute

setProperty (*propertyDict*) → each property in the parameter dictionary is set to the dictionary's value.

Parameters *propertyDict* : {dict}

Dictionary with properties (keyword) and the values that should be set.

See also:

getProperty

setWindowFlags (*flags*) → set window flags of corresponding widget.

The window flags are used to set the type of a widget, dialog or window including further hints to the window system. This method is used to set the entire or-combination of all flags, contained in the Qt-enumeration Qt::WindowType.

The most important types are:

- Qt::Widget (0) -> default type for widgets
- Qt::Window (1) -> the widget looks and behaves like a windows (title bar, window frame...)
- Qt::Dialog (3) -> window decorated as dialog (no minimize or maximize button...)

Further hints can be (among others):

- Qt::FramelessWindowHint (0x00000800) -> borderless window (user cannot move or resize the window)
- Qt::WindowTitleBar (0x00001000) -> gives the window a title bar
- Qt::WindowMinimizeButtonHint (0x00004000) -> adds a minimize button to the title bar
- Qt::WindowMaximizeButtonHint (0x00008000) -> adds a maximize button to the title bar
- Qt::WindowCloseButtonHint (0x00010000) -> adds a close button.

If you simply want to change one hint, get the current set of flags using **getWindowFlags**, change the necessary bitmask and set it again using this method.

Parameters *flags* : {int}

window flags to set (or-combination, see Qt::WindowFlags)

See also:

getWindowFlags

10.7.2 ui-Class

class *itom.ui* (*filename* [, *type*, *dialogButtonBar*, *dialogButtons*, *childOfMainWindow*, *deleteOnClose*, *dockWidgetArea*]) → instance of user interface

Bases: *itom.uiItem*

The class **ui** wraps a user interface, externally designed and given by a ui-file. If your user interface is a dialog or window, chose *ui.TYPEWINDOW* as type, if the user interface is a widget (simplest case), chose *ui.TYPEDIALOG* and your widget will be embedded in a dialog, provided by *itom*. This dialog can be equipped with a button bar, whose buttons are already connected to *itom* internal methods. If you then show your dialog in a modal mode, *itom* knows which button has been clicked in order to accept or reject the dialog.

Parameters *filename* : {str}

path to user interface file (*.ui), absolute or relative to current directory

type : {int}, optional

display type:

- 0 (ui.TYPEDIALOG): ui-file is embedded in auto-created dialog (default),
- 1 (ui.TYPEWINDOW): ui-file is handled as main window,
- 2 (ui.TYPEDOCKWIDGET): ui-file is handled as dock-widget and appended to the main-window dock area

dialogButtonBar : {int}, optional

Only for type ui.TYPEDIALOG (0). Indicates whether buttons should automatically be added to the dialog:

- 0 (ui.BUTTONBAR_NO): do not add any buttons (default)
- 1 (ui.BUTTONBAR_HORIZONTAL): add horizontal button bar
- 2 (ui.BUTTONBAR_VERTICAL): add vertical button bar

dialogButtons : {dict}, optional

every dictionary-entry is one button. key is the role, value is the button text

childOfMainWindow : {bool}, optional

for type TYPEDIALOG and TYPEWINDOW only. Indicates whether window should be a child of itom main window (default: True)

deleteOnClose : {bool}, optional

Indicates whether window should be deleted if user closes it or if it is hidden (default: Hidden, False)

dockWidgetArea : {int}, optional

Only for type ui.TYPEDOCKWIDGET (2). Indicates the position where the dock widget should be placed:

- 1 (ui.LEFTDOCKWIDGETAREA)
- 2 (ui.RIGHTDOCKWIDGETAREA)
- 4 (ui.TOPDOCKWIDGETAREA): default
- 8 (ui.BOTTOMDOCKWIDGETAREA)

static createNewPluginWidget (*widgetName*[, *mandparams*, *optparams*]) → creates widget defined by any algorithm plugin and returns the instance of type 'ui'

Parameters widgetName : {}

name algorithm widget parameters to pass to the plugin. The parameters are parsed and unnamed parameters are used in their incoming order to fill first mandatory parameters and afterwards optional parameters. Parameters may be passed with name as well but after the first named parameter no more unnamed parameters are allowed.

Notes

doctodo

static getDouble (*title*, *label*, *defaultValue*[, *min*, *max*, *decimals=3*]) → shows a dialog to get a double value from the user

Parameters title : {str}

is the dialog title

label : {str}

is the label above the spin box

defaultValue : {double}, optional

is the default value in the spin box

min : {double}, optional

default = -2147483647.0 is the allowed minimal value

max : {double}, optional

default = 2147483647.0 is the allowed maximal value

decimals : {int}, optional

the maximum number of decimal places (default: 1)

Returns out : {tuple (double, bool)}

A tuple where the first value contains the current double value. The second value is True if the dialog has been accepted, else False.

See also:

getInt, *getText*, *getItem*

static getExistingDirectory (*caption*, *startDirectory*[, *options*, *parent*]) → opens a dialog to choose an existing directory

Parameters caption : {str}

is the caption of this dialog

startDirectory : {str}

is the start directory

options : {int}, optional

is an or-combination of the following options (see ‘QFileDialog::Option’):

- 1: ShowDirsOnly [default]
- 2: DontResolveSymlinks
- ... (for others see Qt-Help)

parent : {ui}, optional

is a parent dialog or window, this dialog becomes modal.

Returns out : {str, None}

The selected directory is returned as absolute path or None if the dialog has been rejected.

See also:

getSaveFileName, *getOpenFileName*

static getInt (*title*, *label*, *defaultValue*[, *min*, *max*, *step=1*]) → shows a dialog to get an integer value from the user

Parameters title : {str}

is the dialog title

label : {str}

is the label above the spinbox

defaultValue : {int}, optional

is the default value in the spinbox

min : {int}, optional

is the allowed minimal value (default: -2147483647)

max : {int}, optional

is the allowed maximal value (default: 2147483647)

step : {int}, optional

is the step size if user presses the up/down arrow (default: 1)

Returns out : {tuple (int, bool)}

A tuple where the first value contains the current integer value. The second value is True if the dialog has been accepted, else False.

See also:

getDouble, getText, getItem

static getItem (*title, label, stringList* [, *currentIndex=0, editable=True*]) → shows a dialog to let the user select an item from a string list

Parameters title : {str}

is the dialog title

label : {str}

is the label above the text box

stringList : {tuple or list}, optional

is a list or tuple of possible string values

currentIndex : {int}, optional

defines the preselected value index (default: 0)

editable : {bool}, optional

defines whether new entries can be added (True) or not (False, default)

Returns out : {tuple (str, bool)}

A tuple where the first value contains the current active or typed string value. The second value is True if the dialog has been accepted, else False.

See also:

getInt, getDouble, getText

static getOpenFileName ([*caption, startDirectory, filters, selectedFilterIndex, options, parent*]) → opens dialog for choosing an existing file.

Parameters caption : {str}, optional

This is the optional title of the dialog, default: no title

startDirectory {str}, optional :

optional, if not indicated currentDirectory will be taken

filters : {str}, optional

default = 0 possible filter list, entries should be separated by ;; , e.g. 'Images (.png *.jpg);;Text files (.txt)'

selectedFilterIndex : {int}, optional

is the index of filters which is set by default (0 is first entry)

options : {int}, optional

default = 0 or-combination of enum values QFileDialog::Options

parent : {ui}, optional

is the parent widget of this dialog

Returns out : {str, None}

filename as string or None if dialog has been aborted.

See also:

getSaveFileName

static getSaveFileName ([*caption, startDirectory, filters, selectedFilterIndex, options, parent*])
 → opens dialog for choosing a file to save.

This method creates a modal file dialog to let the user select a file name used for saving a file.

Parameters caption : {str}, optional

This is the title of the dialog

startDirectory : {String}, optional

if not indicated, the current working directory will be taken

filters : {str}, optional

possible filter list, entries should be separated by ;; , e.g. 'Images (.png *.jpg);;Text files (.txt)'

selectedFilterIndex : {int}, optional

default = 0 is the index of filters which is set by default (0 is first entry)

options : {int}, optional

default = 0 or-combination of enum values QFileDialog::Options

parent : {ui}, optional

is the parent widget of this dialog

Returns out : {str, None}

filename as string or None if dialog has been aborted.

See also:

getOpenFileName

static getText (*title, label, defaultString*) → opens a dialog in order to ask the user for a string

Parameters title : {str}

is the dialog title

label : {str}

is the label above the text box

defaultString : {str}

is the default string in the text box

Returns out : {tuple (str, bool)}

A tuple where the first value contains the current string value. The second value is True if the dialog has been accepted, else False.

See also:

getInt, getDouble, getItem

hide () → hides initialized user interface

See also:

show

isVisible () → returns true if dialog is still visible

Returns `visibility` : {bool}

True if user interface is visible, False if it is hidden

static msgCritical (*title*, *text*[, *buttons*, *defaultButton*, *parent*]) → opens a critical message box

Parameters `title` : {str}

is the message box title

`text` : {str}

is the message text

`buttons` : {int}, optional

is an or-combination of `ui.MsgBox[...]`-constants indicating the buttons to display. Use `|` for the or-combination.

`defaultButton` : {int}, optional

is a value of `ui.MsgBox[...]` which indicates the default button

`parent` : {ui}, optional

is the parent dialog of the message box.

See also:

msgWarning, *msgQuestion*, *msgInformation*

static msgInformation (*title*, *text*[, *buttons*, *defaultButton*, *parent*]) → opens an information message box

Parameters `title` : {str}

is the message box title

`text` : {str}

is the message text

`buttons` : {int}, optional

is an or-combination of `ui.MsgBox[...]`-constants indicating the buttons to display. Use `|` for the or-combination.

`defaultButton` : {int}, optional

is a value of `ui.MsgBox[...]` which indicates the default button

`parent` : {ui}, optional

is the parent dialog of the message box.

See also:

msgCritical, *msgQuestion*, *msgWarning*

static msgQuestion (*title*, *text*[, *buttons*, *defaultButton*, *parent*]) → opens a question message box

Parameters `title` : {str}

is the message box title

`text` : {str}

is the message text

`buttons` : {int}, optional

is an or-combination of `ui.MsgBox[...]`-constants indicating the buttons to display. Use `|` for the or-combination.

`defaultButton` : {int}, optional

is a value of `ui.MsgBox[...]` which indicates the default button

parent : {ui}, optional

is the parent dialog of the message box.

See also:

msgCritical, msgWarning, msgInformation

static msgWarning (*title, text*[, *buttons, defaultButton, parent*]) → opens a warning message box

Parameters **title** : {str}

is the message box title

text : {str}

is the message text

buttons : {int}, optional

is an or-combination of `ui.MsgBox[...]`-constants indicating the buttons to display. Use `|` for the or-combination.

defaultButton : {int}, optional

is a value of `ui.MsgBox[...]` which indicates the default button

parent : {ui}, optional

is the parent dialog of the message box.

See also:

msgCritical, msgQuestion, msgInformation

show ([*modal=0*]) → shows initialized UI-Dialog

Parameters **modal** : {int}, optional

- 0: non-modal (default)
- 1: modal (python waits until dialog is hidden)
- 2: modal (python returns immediately)

See also:

hide

10.8 dataObject

class `itom.dataObject` ([*dims* [, *dtype='uint8'* [, *continuous = 0* [, *data = valueOrSequence*]]]]) → constructor to get a new dataObject.

The `itom.dataObject` represents a multidimensional array of fixed-size items with corresponding meta information (units, axes descriptions, scalings, tags, protocol...). Recently the following data types (*dtype*) are supported:

- Integer-type (`int8, uint8, int16, uint16, int32, uint32`),
- Floating-type (`float32, float64` (=> double)),
- Complex-type (`complex64` (2x `float32`), `complex128` (2x `float64`)).
- Color-type (`rgba32` (`uint32` or `uint[4]` containing the four 8bit values [R, G, B, Alpha])).

Arrays can also be constructed using some of the static pre-initialization methods ‘zeros’, ‘ones’, ‘rand’ or ‘randN’ (refer to the See Also section below).

Parameters **dims** : {sequence of integers}, optional

'dims' is a list or tuple indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns. If not given, an empty data object is created.

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8', 'uint8', ..., 'int32', 'uint32', 'float32', 'float64', 'complex64', 'complex128', 'rgba32'

continuous : {int}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

data : {str}, optional

'data' is a single value or a sequence with the same amount of values than the data object. The values from data will be assigned to the new data object (filled row by row).

See also:

[*ones*](#), [*zeros*](#), [*rand*](#), [*randN*](#)

Notes

The `itom.dataObject` is a direct wrapper for the underlying C++ class *dataObject*. This array class mainly is based on the class *Mat* of the computer vision library (OpenCV).

In order to handle huge matrices, the data object can divide one array into chunks in memory. Each subpart (called matrix-plane) is two-dimensional and covers data of the last two dimensions. In c++-context each of these matrix-planes is of type `cv::Mat_<type>` and can be used with every operator given by the openCV-framework (version 2.3.1 or higher).

The dimensions of the matrix are structured descending. So if we assume to have a n-dimensional matrix A, where each dimension has its size `s_i`, the dimensions order is n, ..., z, y, x and the corresponding sizes of A are `[s_n, s_(n-1), s_(n-2), ..., s_y, s_x]`.

In order to make the data object compatible to continuously organized data structures, like numpy-arrays, it is also possible to have all matrix-planes in one data-block in memory (not recommended for huge matrices). Nevertheless, the indicated data structure with the two-dimensional sub-matrix-planes is still existing. The data organization is equal to the one of openCV, hence, two-dimensional matrices are stored row-by-row (C-style)...

In addition to OpenCV, `itom.dataObject` supports complex valued data types for all operators and methods.

Warning 'uint32' is not fully openCV-compatible and hence causes instability!

Deep Copy, Shallow Copy and ROI

It is possible to set a n-dimensional region of interest (ROI) to each matrix, the virtual dimensions, which will be delivered if the user asks for the matrix size. To avoid copy operations where possible a simple `=_Operator` will also make a shallow copy of the object. Shallow copies share the same data (elements and meta data) with the original object, hence manipulations of one object will affect the original object and all shallow copies.

The opposite a deep copy of a `dataObject` (by `sourceObject.copy()`) creates a complete new matrix with own meta data object.

Example:

```
#Create an object
dObj = dataObject([5, 10, 10], 'int8')

# Make a shallow copy
dObjShallow = dObj

# Make a shallow copy on ROI
```



```

dObjROI = dObj[1, :, :]

# Set the value of element [1, 0, 0] to 0
dObj[1, 0, 0] = 0

# Make a deep copy of the dObjROI
dObjROICopy = dObjROI.copy()

# Set the value of dObjROICopy element [0, 0, 0] to 127 without effecting other objects
dObjROICopy[0, 0, 0] = 127

```

Constructor

The function `dataObject([dims [, dtype='uint8', continuous = 0][, data = valueOrSequence]])` creates a new itom-dataObject filled with undefined data. If no parameters are given, an uninitialized DataObject (dims = 0, no sizes) is created.

As second possibility you can also use the copy-constructor '`dataObject(AnyArray)`', where AnyArray must be any array-like structure which is parsable by the numpy-interface.

abs() → return a new data object with the absolute values of the source

This method calculates the abs value of each element in source and writes the result to the output object. In case of floating point or real object, the type of the output will not change. For complex values the type is changes to the corresponding floating type value.

Returns **res** : {dataObject}

output dataObject of same shape but the type may be changed.

addToProtocol(newLine) → Appends a protocol line to the protocol.

Appends a line of text to the protocol string of this data object. If this data object has got a region of interest defined, the rectangle of the ROI is automatically appended to newLine. The protocol string ends with a newline character.

Address the content of the protocol by `obj.tags["protocol"]`. The protocol is contained in the ordinary tag dictionary of this data object under the key 'protocol'.

Parameters **newLine** : {str}

The text to be added to the protocol.

adj() → Adjugate all elements

Every plane (spanned by the last two axes) is transposed and every element is replaced by its complex conjugate value.

Raises **TypeError** :

if data type of this data object is not complex.

See also:

adjugate

adjugate() → returns the plane-wise adjugated array of this dataObject.

If this data object has a complex type, the tranposed data object is returned where every element is complex conjugated. For data objects with more than two dimensions the tranposition is done plane-wise, hence, only the last two dimensions are permutated.

Returns **out** : {dataObject}

adjugate of this dataObject

Raises **TypeError** :

if data type of this data object is not complex.

See also:

adj

adjustROI (*offsetList*) → adjust the size and position of the region of interest of this data object

For every data object, it is possible to define a region of interest such that subsequent commands only refer to this subpart. However, if values within the region of interest (ROI) are changed, this also affects the original data object due to the shallow copy principal of python. Use this command to adjust the current size and position of this region of interest by passing an offset list, that contains integer numbers with twice the size than the number of dimensions.

Example:

```
d = dataObject([5,4])
droi = d
droi.adjustROI([-2,0,-1,-1])
```

Now *droi* is a region of interest of the original data object whose first value is equal to *d*[2,1] and its size is (3,2)

Parameters *offsetList* : {list of integers}

This list must have twice as many values than the number of dimensions of this data object. A pair of numbers indicates the shift of the current boundaries of the region of interest in every dimension. The first value of each pair is the offset of the ‘left’ boundary, the second the shift of the right boundary. A positive value means a growth of the region of interest, a negative one let the region of interest shrink towards the center.

See also:

locateROI

arg () → return a new data object with the argument values of the source

This method calculates the argument value of each element in source and writes the result to the output object. This object must be of complex type (complex128 or complex64). The output value will be float type (float64 or float32).

Returns *res* : {dataObject}

output dataObject of same shape but the type is changed.

astype (*typestring*) → converts this data object to another type

Converts this data object to a new data object with another type, given by the string *newTypestring* (e.g. ‘uint8’). The converted data object is a deep copy of this object if the new type does not correspond to the current type, else a shallow copy of this object is returned.

Parameters *typestring* : {str}

Type string indicating the new type (‘uint8’,...,‘float32’,...,‘complex64’)

Returns *c* : {dataObject}

type-converted data object

Notes

This method mainly uses the method *convertTo* of OpenCV.

conj () → complex-conjugates all elements of this dataObject (inline).

Every value of this dataObject is replaced by its complex-conjugate value.

Raises **TypeError** :

if data type of this data object is not complex.

See also:

conjugate

conjugate () → return a copy of this dataObject where every element is complex-conjugated.

Returns *out* : {dataObject}

element-wise complex conjugate of this data object

Raises `TypeError` :

if data type of this data object is not complex.

See also:

conj

copy (*regionOnly=0*) → return a deep copy of this dataObject

Parameters *regionOnly* : {bool}, optional

If *regionOnly* is 1, only the current region of interest of this dataObject is copied, else the entire dataObject including the current settings concerning the region of interest are deeply copied [default].

Returns *cpy* : {dataObject}

Deep copy of this dataObject

data () → prints the content of the dataObject in a readable form.

Notes

When calling this method, the complete content of the dataObject is printed to the standard output stream.

deleteTag (*key*) → Delete a tag specified by key from the tag dictionary.

Checks whether a tag with the given key exists in the tag dictionary and if so deletes it.

Parameters *key* : {str}

the name of the tag to be deleted

Returns *success* : {bool}:

True if tag with given key existed and could be deleted, else False

div (*obj*) → a.div(b) return result of element wise division of a./b

All meta information (axis scales, offsets, descriptions, units, tags...) of the resulting object are copied from this data object.

Parameters *obj* : {dataObject}

Every value in this data object is divided by the corresponding value in obj.

Returns *c* : {dataObject}

Resulting divided data object.

existTag (*key*) → return True if tag with given key exists, else False

Checks whether a tag with the given key exists in tag dictionary of this data object and returns True if such a tag exists, else False.

Parameters *key* : {str}

the key of the tag

Returns *result* : {bool}

True if tag exists, else False

static eye (*size* [, *dtype='uint8'*]) → creates a 2D, square, eye-matrix.

Static method for creating a two-dimensional, square, eye-matrix of type itom.dataObject.

Parameters *size* : {int},

the size of the square matrix (single value)

dtype : {str}, optional

'dtype' is the data type of each element, possible values:
'int8','uint8',..., 'int32','uint32','float32','float64','complex64','complex128'

Returns **I** : {dataObject} of shape (size,size)

An array where all elements are equal to zero, except for the 'k'-th diagonal, whose values are equal to one.

See also:

ones method for creating a matrix filled with ones

zeros method for creating a matrix filled with zeros

getTagListSize () → returns the number of tags in the tag dictionary

Every data object can have an arbitrary number of tags stored in the tag dictionary. This method returns the number of different tags, where the protocol is also one tag with the key 'protocol'.

Returns **length** : {int}:

size of the tag dictionary. The optional protocol also counts as one item.

imag () → return a new data object with the imaginary part of the source

This method extracts the imaginary part of each element in source and writes the result to the output object. This object must be of complex type (complex128 or complex64). The output value will be float type (float64 or float32).

Returns **res** : {dataObject}

output dataObject of same shape but the type is changed.

locateROI () → returns information about the current region of interest of this data object

A region of interest (ROI) of a data object is defined by the two values per axis. The first element always indicates the size between the real border of the data object and the region of interest on the left / top ... side and the second value the margin of the right / bottom ... side.

This method returns a tuple with two elements: The first is a list with the original sizes of this data object, the second is a list with the offsets from the original data object to the first value in the current region of interest

If no region of interest is set (hence: full region of interest), the first list corresponds to the one returned by size(), the second list is a zero-vector.

See also:

adjustROI

makeContinuous () → return continuous representation of dataObject

Per default a dataObject with more than two dimensions allocates separated chunks of memory for every plane, where a plane is always the matrix given by the last two dimensions. This separated storage usually allows allocating more memory for huge for instance three dimensional matrices. However, in order to generate a dataObject that is directly compatible to Numpy or other C-style matrix structures, the entire allocated memory must be in one block, that is called continuous. If you create a Numpy array from a dataObject that is not continuous, this function is implicitly called in order to firstly make the dataObject continuous before passing to Numpy.

Returns **obj** : {dataObject}

continuous dataObject

Notes

if this dataObject already is continuous, a simple shallow copy is returned

mul (*obj*) → a.mul(b) returns element wise multiplication of a*b

All meta information (axis scales, offsets, descriptions, units, tags...) of the resulting object are copied from this data object.

Parameters **obj** : {dataObject}

dataObject whose values are element-wisely multiplied with the values in this dataObject.

Returns **c** : {dataObject}

Resulting multiplied data object.

For a mathematical multiplication see the *-operator. :

name () -> returns the name of this object (dataObject)

normalize ([*minValue*=0.0, *maxValue*=1.0, *typestring*='']) → returns the normalization of this dataObject

Returns the normalized version of this data object, where the values lie in the range [min-Value,maxValue]. Additionally it is also possible to convert the resulting data object to another type (given by the parameter typestring). As default no type conversion is executed.

Parameters **minValue** : {double}

minimum value of the normalized range

maxValue : {double}

maximum value of the normalized range

typestring : {String}

Type string indicating the new type ('uint8',...,'float32',...,'complex64'), default: '' (no type conversion)

Returns **normalized** : {dataObject}

normalized data object

static ones (*dims*[, *dtype*='uint8'[, *continuous* = 0]]) → creates new dataObject filled with ones.

Static method for creating a new n-dimensional itom.dataObject with given number of dimensions and dtype, filled with ones.

Parameters **dims** : {integer list}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8','uint8',...,'int32','float32','float64','complex64','complex128','rgba32'

continuous : {int}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns **I** : {dataObject} of shape (size,size)

An array where all elements are equal to one.

See also:

[eye](#) method for creating an eye matrix

[zeros](#) method for creating a matrix filled with zeros

Notes

For color-types (rgba32) every item / cell will be white: [r=255 g=255 b=255 a=255].

physToPix (*values* [, *axes*]) → returns the pixel coordinates for the given physical coordinates.

This method transforms a physical axis coordinate into its corresponding pixel coordinate. The transformation is influenced by the offset and scaling of each axis:

$\text{phys} = (\text{pix} - \text{offset}) * \text{scaling}$

If no axes parameter is given, the values are assumed to belong the the ascending axis list (0,1,2,3...).

Parameters values : {float, float-tuple}

One single physical coordinate or a tuple of physical coordinates.

axes : {int, int-tuple}, optional

If this is given, the values are mapped to the axis indices given by this value or tuple.
Else, an ascending list starting with index 0 is assumed.

Returns **Float or float-tuple with the pixel coordinates for each physical coordinate at the given axis index.** :

Raises **Value error** : :

if the given axes is invalid (out of range)

pixToPhys (*values* [, *axes*]) → returns the physical coordinates for the given pixel coordinates.

This method transforms a pixel coordinate into its corresponding physical coordinate. The transformation is influenced by the offset and scaling of each axis:

$\text{pix} = (\text{phys} / \text{scaling}) + \text{offset}$

If no axes parameter is given, the values are assumed to belong the the ascending axis list (0,1,2,3...).

Parameters values : {float, float-tuple}

One single pixel coordinate or a tuple of pixel coordinates.

axes : {int, int-tuple}, optional

If this is given, the values are mapped to the axis indices given by this value or tuple.
Else, an ascending list starting with index 0 is assumed.

Returns **Float or float-tuple with the physical coordinates for each pixel coordinate at the given axis index.** :

Raises **Value error** : :

if the given axes is invalid (out of range)

static rand ([*dims* [, *dtype*='uint8' [, *continuous* = 0]]]) → creates new dataObject filled with uniform distributed random values.

Static method to create a new itom.dataObject filled with uniform distributed random numbers. In case of an integer type, the uniform noise is from min<ObjectType>(inclusiv) to max<ObjectType>(inclusiv). For floating point types, the noise is between 0(inclusiv) and 1(exclusiv).

Parameters dims : {integer list}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns.

dtype : {str}, optional

'dtype' is the data type of each element, possible values:
'int8', 'uint8', ..., 'int32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {int}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns out : {dataObject}

Array of random numbers with the given dimensions, dtype.

See also:

[*randN*](#) method for creating a matrix filled with gaussianly distributed values

static randN (*dims*[, *dtype*='uint8'[, *continuous* = 0]]) → creates dataObject filled with gaussian distributed random values.
 Static method to create a new itom.dataObject filled with gaussian distributed random numbers. In case of an integer type, the gaussian noise mean value is (max+min)/2.0 and the standard deviation is (max-min)/6.0 to max. For floating point types, the noise mean value is 0 and the standard deviation is 1.0/3.0.

Parameters dims : {integer list}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns.

dtype : {str}, optional

'dtype' is the data type of each element, possible values: 'int8','uint8',..., 'int32', 'float32', 'float64', 'complex64', 'complex128'

continuous : {int}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns out : {dataObject}

Array of random numbers with the given dimensions, dtype.

See also:

[*rand*](#) method for creating a matrix filled with uniformly distributed values

real () → return a new data object with the real part of the source
 This method extracts the real part of each element in source and writes the result to the output object. This object must be of complex type (complex128 or complex64). The output value will be float type (float64 or float32).

Returns res : {dataObject}

output dataObject of same shape but the type is changed.

reshape (*newSizes*) → Returns reshaped shallow copy of data object

Notes

Not implemented yet.

setAxisDescription (*axisNum*, *axisDescription*) → Set the description of the specified axis.
 Each axis in the data object can get a specific axisDescription string (e.g. mm). Use this method to set the axisDescription of one specific axis.

Parameters axisNum : {int}

The addressed axis index

axisDescription : {str}

New axis description

Raises Runtime error :

if the given axisNum is invalid (out of range)

See also:

axisDescriptions this attribute can directly be used to read/write the axis description(s) of single or all axes

setAxisOffset (*axisNum*, *axisOffset*) → Set the offset of the specified axis.

Each axis in the data object can get a specific scale value, described in axisUnits per pixel. Use this method to set the scale of one specific axis. The value of each pixel in its physical unit is the (px-Coordinate - axisOffset) * axisScale

Parameters **axisNum** : {int}

The addressed axis index

axisOffset : {double}

New axis offset in [px]

Raises **Runtime error** : :

if the given axisNum is invalid (out of range)

See also:

axisOffsets this attribute can directly be used to read/write the axis offset(s) of single or all axes

setAxisScale (*axisNum*, *axisScale*) → Set the scale value of the specified axis.

Each axis in the data object can get a specific scale value, described in axisUnits per pixel. Use this method to set the scale of one specific axis.

Parameters **axisNum** : {int}

The addressed axis index

axisScale : {double}

New axis scale in axisUnit/px

Raises **Runtime error** : :

if the given axisNum is invalid (out of range)

See also:

axisScales this attribute can directly be used to read/write the axis scale(s) of single or all axes

setAxisUnit (*axisNum*, *axisUnit*) → Set the unit of the specified axis.

Each axis in the data object can get a specific unit string (e.g. mm). Use this method to set the unit of one specific axis.

Parameters **axisNum** : {int}

The addressed axis index

axisUnit : {str}

New axis unit

Raises **Runtime error** : :

if the given axisNum is invalid (out of range)

See also:

axisUnits this attribute can directly be used to read/write the axis unit(s) of single or all axes

setTag (*key*, *tagvalue*) → Set the value of tag specified by key.

Sets the value of an existing tag (defined by key) in the tag dictionary to the string or double tagvalue or adds a new item with key.

Parameters **key** : {str}

the name of the tag to set

tagvalue : {str or double}

the new value of the tag, either string or double value

Notes

Do NOT use 'special character' within the tag key because they are not XML-save.

size ([*index*]) → returns the size of this dataObject (tuple of the sizes in all dimensions or size in dimension indicated by optional axis index).

Parameters **index** : {int}, optional

If index is given, only the size of the indicated dimension is returned as single number (0 ≤ index < number of dimensions)

Returns A tuple containing the sizes of all dimensions or one single size value if 'index' is indicated. :

See also:

shape the read-only attribute shape is equal to size()

Notes

For a more consistent syntax with respect to numpy arrays, the same result is obtained by the attribute shape. Please use the attribute shape for future implementations since this method is marked as deprecated.

squeeze () → return a squeezed shallow copy (if possible) of this dataObject.

This method removes every dimension with size equal to 1. Take care, that none of the last two dimensions is considered by this squeeze-command.

Returns **squeezed** : {dataObject}

The squeezed data object where all kept planes are shallow copies of the original plane.

Notes

The returned squeezed data object is a shallow copy of the original data object and hence changes in its values will also change the original data set. This method is equal to numpy.squeeze

toGray ([*destinationType*='uint8']) → returns the rgba32 color data object as a gray-scale object

The destination data object has the same size than this data object and the real type given by destinationType. The pixel-wise conversion is done using the formula: gray = 0.299 * red + 0.587 * green + 0.114 * blue. Parameters ——— destinationType : {str}

Type string indicating the new real type ('uint8',... 'float32', 'float64' - no complex)

Returns **dataObj** : {dataObject}

converted gray-scale data object of desired type

tolist () → return the data object as a (possibly nested) list

This method returns a nested list with all values of this data object. The recursion level of this nested list corresponds to the number of dimensions. The outer list corresponds to the first dimension.

Returns **y** : {list}

Nested list with values of data object (int, float or complex depending on type of data object)

trans () → return a plane-wise transposed dataObject

Return a new data object with the same data type than this object and where every plane (data spanned by the last two dimensions) is transposed respectively such that the last two axes are permuted.

Returns out : {dataObject}

A copy of this dataObject is returned where every plane is its transposed plane.

static zeros (*dims* [, *dtype*='uint8' [, *continuous* = 0]]) → creates new dataObject filled with zeros.

Static method for creating a new n-dimensional itom.dataObject with given number of dimensions and dtype, filled with zeros.

Parameters dims : {integer list}

'dims' is list indicating the size of each dimension, e.g. [2,3] is a matrix with 2 rows and 3 columns

dtype : {str}, optional

'dtype' is the data type of each element, possible values:
'int8', 'uint8', ..., 'int32', 'float32', 'float64', 'complex64', 'complex128', 'rgba32'

continuous : {int}, optional

'continuous' [0|1] defines whether the data block should be continuously allocated in memory [1] or in different smaller blocks [0] (recommended for huge matrices).

Returns I : {dataObject} of shape (size,size)

An array where all elements are equal to zero.

See also:

[**eye**](#) method for creating an eye matrix

[**ones**](#) method for creating a matrix filled with ones

Notes

For color-types (rgba32) every item / cell will be black and transparent: [r=0 g=0 b=0 a=0].

axisDescriptions

tuple containing the axis descriptions {str}.

This attribute gives access to the internal axis descriptions expressed as a tuple of strings. The tuple has the same length than the number of dimensions of this data object.

You can either assign a new tuple with the same length or change single values using tuple indexing.

See also:

setAxisDescriptions alternative method to change the description string of one single axis

Notes

read / write

axisOffsets

tuple containing the axis offsets [px].

This attribute gives access to the internal axis offsets [px] expressed as a tuple of double values. The i-th value in the tuple corresponds to the pixel-offset of the i-th axis. Either assign a new tuple with the same length than the number of dimensions or change single values using tuple indexing.

Definition: Physical unit = (px-Coordinate - offset)* scale

See also:

setAxisOffset

Notes

read / write

axisScales

tuple containing the axis scales [unit/px].

This attribute gives access to the internal axis scales [unit/px] expressed as a tuple of double values. The i-th value in the tuple corresponds to the scaling factor of the i-th axis. Either assign a new tuple with the same length than the number of dimensions or change single values using tuple indexing.

Definition: Physical unit = (px-Coordinate - offset)* scale

See also:

setAxisScale

Notes

read / write

axisUnits

tuple containing the axis units {str}.

This attribute gives access to the internal axis units expressed as a tuple of strings. The tuple has the same length than the number of dimensions of this data object.

You can either assign a new tuple with the same length or change single values using tuple indexing.

See also:

setAxisUnits alternative method to change the unit string of one single axis

Notes

read / write

base

base object

continuous

true if matrix is continuously organized, else false.

If true, the whole matrix is allocated in one huge block in memory, hence, this data object can be transformed into a numpy representation (npDataObject) without reallocating memory.

Notes

read-only

dims

number of dimensions of this data object

Notes

read-only property, this property is readable both by the attributes `ndim` and `dims`.

dtype

get type string of data in this data object

This type string has one of these values: `'uint8'`, `'int8'`, `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'float32'`, `'float64'`, `'complex64'`, `'complex128'`, `'rgba32'`

Notes

This attribute is read-only

metaDict

return dictionary with all meta information of this dataObject

Returns a new dictionary with the following meta information:

- `axisOffsets` : List with offsets of each axis
- `axisScales` : List with the scales of each axis
- `axisUnits` : List with the unit strings of each axis
- `axisDescriptions` : List with the description strings of each axis
- `tags` : Dictionary with all tags including the tag `'protocol'` if at least one protocol entry has been added using `addToProtocol`
- `valueOffset` : Offset of each value (0.0)
- `valueScale` : Scale of each value (1.0)
- `valueDescription` : Description of the values
- `valueUnit` : The unit string of the values

Notes

Adding or changing values to `/` in the dictionary does not change the meta information of the dataObject. Use the corresponding setters like `setTag...` instead.

ndim

number of dimensions of this data object

Notes

read-only property, this property is readable both by the attributes `ndim` and `dims`.

shape

tuple with the sizes of each dimension / axis of this data object.

See also:

size

Notes

In difference to the `shape` attribute of numpy arrays, no new shape tuple can be assigned to this value (used to `'reshape'` the array). Read-only.

tags

tag dictionary of this data object.

This attribute returns a dict_proxy object of the tag dictionary of this data object. This object is read-only. However you can assign an entire new dictionary to this attribute that fully replaces the old tag dictionary. The tag dictionary can contain arbitrary pairs of key -> value where value is either a string or a double value.

Special tags are the key 'protocol' that contains the newline-separated protocol string of the data object (see: addToProtocol()) or the key 'title' that can for instance be used as title in any plots.

You can add single elements using the method setTag(key,value) or you can delete tags using deleteTag(key).

Do NOT use 'special character' within the tag key because they are not XML-save.

Notes

read-only / write only for fully new dictionary

value

get/set the values within the ROI as a one-dimensional tuple.

This method gets or sets the values within the ROI. If this attribute is called by means of a getter, a tuple is returned which is created by iterating through the values of the data object (row-wise). In the same way of iterating, the values are set to the data object if you provide a tuple of the size of the data object or its ROI, respectively.

Example:

```
b = dataObject[1,1:10,1,1].value
# or for the first value
b = dataObject[1,1:10,1,1].value[0]
# The elements of the tuple are adressed with b[idx].
```

valueDescription

value unit description.

Attribute to read or write the unit description string of the values in this data object.

Notes

read / write

valueOffset

value offset [default: 0.0].

This attribute gives the offset of each value in the data object. This value is always 0.0.

Notes

This attribute is read only

valueScale

value scale [default: 0.0].

This attribute gives the scaling factor of each value in the data object. This value is always 1.0.

Notes

This attribute is read only

valueUnit

value unit.

Attribute to read or write the unit string of the values in this data object.

Notes

read / write

xyRotationalMatrix

Access the 3x3 rotational matrix in the dataObject tag space

This attribute gives access to the xyRotationalMatrix in the metaData-Tag space. The getter method returns a 3x3-Array deep copied from the internal matrix, Implemented to offer compability to x3p format.

Notes

{3x3 array of doubles} : ReadWrite

10.9 point

class itom.point (*[type, [xyz, [intensity,][rgba,][normal,][curvature]]*) → creates new point used for class 'pointCloud'.

Parameters *type* : {int}

the desired type of this point (default: point.PointInvalid). Depending on the type, some of the following parameters must be given:

xyz : {seq}, all types besides PointInvalid

sequence with three floating point elements (x,y,z)

intensity : {float}, only PointXYZI or PointXYZINormal

is a floating point value for the intensity

rgba, {seq. of uint8, three or four values}, only PointXYZRGBA or PointXYZRGBNormal :

a uint8-sequence with either three or four values (r,g,b,a). If alpha value is not given, 255 is assumed

normal : {seq}, only PointXYZNormal, PointXYZINormal and PointXYZRGBNormal

is a sequence with three floating point elements (nx, ny, nz)

curvature : {float}, only PointXYZNormal, PointXYZINormal and PointXYZRGBNormal

is the curvature value for the normal (float)

name ()

PointInvalid = 0

PointXYZ = 1

PointXYZI = 2

PointXYZINormal = 16

PointXYZNormal = 8

PointXYZRGBA = 4

PointXYZRGBNormal = 32

curvature

gets or sets curvature value

Notes

{float} : ReadWrite

intensity

gets or sets intensity

Notes

{float} : ReadWrite

normal

gets or sets normal vector (nx,ny,nz)

Notes

{float-list} : ReadWrite

rgb

gets or sets rgb-values (r,g,b)

Notes

{uint8-list} : ReadWrite

rgba

gets or sets rgba-values (r,g,b,a)

Notes

{uint8-list} : ReadWrite

type

returns type-object for this point

xyz

gets or sets x,y,z-values (x,y,z)

Notes

{float-list} : ReadWrite

10.10 pointCloud

class `itom.pointCloud` (*[type]* | *pointCloud* [*,indices*] | *width*, *height* [*,point*] | *point*) → creates new point cloud.

Parameters **type** : point-type (e.g. `point.PointXYZ`, see Notes)

pointCloud : {pointCloud}

another pointCloud instance which is appended at this point cloud

indices : {sequence}, optional

only the indices of the given pointCloud will be appended to this point cloud

width : {int}

width of the new point cloud

height : {int}

height of the new point cloud (the point cloud is dense if `height > 1`)

point : {point}

single point instance. This point cloud is filled up with elements of this point. If width and height are given, every element is set to this point, else the point cloud only consists of this single point.

Notes

Possible types:

- `point.PointXYZ`
- `point.PointXYZI`
- `point.PointXYZRGBA`
- `point.PointXYZNormal`
- `point.PointXYZINormal`
- `point.PointXYZRGBNormal`

append (*point*) → appends point at the end of the point cloud.

Parameters **point** : {point???}, optional

Notes

The type of point must fit to the type of the point cloud. If the point cloud is invalid, its type is set to the type of the point.

clear () → clears the whole point cloud

copy () → returns a deep copy of this point cloud.

Returns **cloud** : {pointCloud}

An exact copy of this point cloud.

dense

specifies if all the data in points is finite (true), or whether it might contain Inf/NaN values (false).

Notes

{bool} : ReadOnly ReadWrite

empty

returns whether this point cloud is empty, hence size == 1

Notes

{ } : ReadOnly ReadWrite

erase (*indices*) -> erases the points in point clouds indicated by indices (single number of slice)

Parameters *indices* : { }

Notes

This method is the same than command 'del pointCloudVariable[indices]'

fields

get available field names of point cloud

Notes

{ } : ReadOnly ReadWrite

static fromDisparity (*disparity* [, *intensity*] [, *deleteNaN*]) → creates a point cloud from a given disparity dataObject.

Creates a point cloud from the 2.5D data set given by the disparity dataObject. The x and y-components of each point are taken from the regular grid values of 'disparity' (considering the scaling and offset of the object). The corresponding z-value is the disparity's value itself.

Parameters *disparity* : {MxN data object, float32}

The values of this dataObject represent the disparity values.

intensity : {MxN data object, float32}, optional

If given, an XYZI-point cloud is created whose intensity values are determined by this dataObject (cannot be used together with 'color')

deleteNaN : {bool}, optional

If true (default: false), NaN or Inf-values (z) in the disparity map will not be copied into the point cloud (the point cloud is not organized any more).

color : {MxN data object, rgba32}, optional

If given, a XYZRGBA-point cloud is created whose color values are determined by this dataObject (cannot be used together with 'intensity')

Returns PointCloud. :

static fromXYZ (*X*, *Y*, *Z* | *XYZ*) → creates a point cloud from three X,Y,Z data objects or from one 3xMxN data object

The created point cloud is not organized (height=1) and dense, if no NaN or Inf values are within the point cloud (deleteNaN:true results in a dense point cloud)

Parameters *X,Y,Z* : {MxN data objects}

Three 2D data objects with the same size.

XYZ : {3xMxN data object}

OR: 3xMxN data object, such that the first plane is X, the second is Y and the third is Z

deleteNaN : {bool}

default = false if True all NaN values are skipped, hence, the resulting point cloud is not dense any more

Returns PointCloud. :

static fromXYZI (*X, Y, Z, I | XYZ, I*) → creates a point cloud from four X,Y,Z,I data objects or from one 3xMxN data object and one intensity data object

The created point cloud is not organized (height=1) and dense, if no NaN or Inf values are within the point cloud (deleteNaN:true results in a dense point cloud)

Parameters X,Y,Z,I : {MxN data objects}

Four 2D data objects with the same size.

OR :

XYZ : {3xMxN data object}

3xMxN data object, such that the first plane is X, the second is Y and the third is Z

I : {MxN data object}

MxN data object with the same size than the single X,Y or Z planes

deleteNaN : {bool}

default = false if True all NaN values are skipped, hence, the resulting point cloud is not dense any more

Returns PointCloud. :

static fromXYZRGBA (*X, Y, Z, color | XYZ, color*) → creates a point cloud from four X,Y,Z,color data objects or from one 3xMxN data object and one coloured data object

The created point cloud is not organized (height=1) and dense, if no NaN or Inf values are within the point cloud (deleteNaN:true results in a dense point cloud)

Parameters X,Y,Z,color : {MxN data objects}

Four 2D data objects with the same size. X,Y,Z must have the same type, color must be rgba32.

OR :

XYZ : {3xMxN data object}

3xMxN data object, such that the first plane is X, the second is Y and the third is Z

color : {MxN data object}

MxN data object with the same size than the single X,Y or Z planes, type: rgba32

deleteNaN : {bool}

default = false if True all NaN or Inf values are skipped, hence, the resulting point cloud does not contain this values

Returns PointCloud. :

height

returns height of point cloud if organized as regular grid (organized == true), else 1 specifies the height of the point cloud dataset in the number of points. HEIGHT has two meanings:

- it can specify the height (total number of rows) of an organized point cloud dataset;
- it is set to 1 for unorganized datasets (thus used to check whether a dataset is organized or not).

Notes

{ } : ReadOnly ReadWrite

insert (*index, values*) → inserts a single point or a sequence of points before position given by index

Parameters **index** : { }

values : { }

Notes

doctodo

name ()

organized

returns whether this point cloud is organized as regular grid, hence height != 1

Notes

{ } : ReadOnly ReadWrite

size

returns number of points in point cloud

Notes

{ } : ReadOnly ReadWrite

toDataObject () → returns a PxN data object, where P is determined by the point type in the point cloud. N is the number of points.

The output has at least 3 elements per column. Onr got each coordinate (xyz). These will always be the first 3 elements. If the pointcloud type has normals defined, these will be added in the next 4 columns as [nx, ny, nz, curvature]. If the pointcloud type has an intensity, these will be added in the next 1 column as [intensity]. If the pointcloud type has an RGB(A)-intensity, these will be added in the next 4 column as [r, g, b, a]. Hence following combinations are possible [x,y,z], [x,y,z,i], [x,y,z,r,g,b,a], [x,y,z,nx,ny,nz, curvature], [x,y,z,nx,ny,nz, curvature, i], ...

Returns **dObj** : {dataObject}

A dataObject with P (cols) by N elements (Points), where the elements per column depend on the point cloud type

Notes

doctodo

type

returns point type of point cloud

Notes

{ } : ReadOnly ReadWrite

width

returns width of point cloud if organized as regular grid (organized == true), else equal to size specifies the width of the point cloud dataset in the number of points. WIDTH has two meanings:

- it can specify the total number of points in the cloud (equal with POINTS see below) for unorganized datasets;
- it can specify the width (total number of points in a row) of an organized point cloud dataset.

Notes

{ } : ReadOnly ReadWrite

10.11 polygonMesh

class `itom.polygonMesh` (*[mesh, polygons]*) → creates a polygon mesh.

This constructor either creates an empty polygon mesh, a shallow copy of another polygon mesh (mesh parameter only) or a deep copy of another polygon mesh where only the polygons, given by the list of indices in the parameter 'polygons', are taken. In this case, the containing cloud is reduced and no longer organized (height=1, dense=false)

Parameters **mesh** : {polygonMesh}, optional

another polygon mesh instance (shallow or deep copy depending on polygons-parameter)

polygons : {sequence or array-like}, optional

If given, polygons must be a sequence or one-dimensional array-like structure, where all values can be transformed into unsigned integer values. Polygons must contain a list of indices pointing to all polygon from the given mesh that should be copied to this new instance.

data ()

prints content of polygon mesh

static **fromCloudAndPolygons** (*cloud, polygons*) → creates a polygon mesh from cloud and polygons.

Parameters **cloud** : {pointCloud}

the input point cloud

polygons : {array-like, MxN}

an array-like matrix with the indices of the polygons. The array contains M polygons and every row gives the indices of the vertices of the cloud belonging to the polygon.

static **fromOrganizedCloud** (*cloud*) → creates a polygon mesh from an organized cloud using triangles.

The polygons are created as triangles. Triangles are also created for non-finite points.

Parameters **cloud** : {pointCloud}

the input point cloud (must be organized, see attribute organized of a cloud)

getCloud (*pointType = point.PointInvalid*) → returns the point cloud of this polygon mesh converted to the desired type.

If the pointType is not given or point.PointInvalid, the type of the internal pointCloud is guessed with respect to available types.

Parameters **pointType** : {int, enum point.PointXXX}, optional

the point type value of the desired type, the point cloud should be converted too (default: point.PointInvalid)

getPolygons ()

name ()

nrOfPolygons

returns the number of polygons in this mesh

10.12 region

class `itom.region` (`[x, y, w, h[, type=region.RECTANGLE]]`) → creates a rectangular or elliptical region.

Bases: `object`

This class is a wrapper for the class `QRegion` of Qt. It provides possibilities for creating pixel-based regions. Furtherone you can calculate new regions based on the intersection, union or subtraction of other regions. Based on the region it is possible to get a `uint8` masked dataObject, where every point within the entire region has the value 255 and all other values 0

Parameters `x` : {int}

x-coordinate of the reference corner of the region

`y` : {int}

y-coordinate of the reference corner of the region

`w` : {int}

width of the region

`h` : {int}

height of the region

type : {int}, optional

`region.RECTANGLE` creates a rectangular region (default). `region.ELLIPSE` creates an elliptical region, which is placed inside of the given boundaries.

Notes

It is also possible to create an empty instance of the region.

10.13 rgba

class `itom.rgba` (`r, g, b[, alpha=255]`) → creates a new color value from red, green, blue and optional alpha

Parameters `r` : {uint8}

red component [0,255]

`g` : {uint8},

green component [0,255]

`b` : {uint8}

blue component [0,255]

alpha : {uint8}, optional

alpha component [0,255], default: 255 (no transparency)

Notes

For a gray value set all colors to the same value.

name ()

alpha

b
blue

g
green

r
red

10.14 autoInterval

class `itom.autoInterval` (`[min=-inf, max=inf, auto=false]`) → creates a new auto interval object.

Parameters `min` : {float}

minimum value of interval (default: -infinity)

max : {float},

maximum value of interval (default: +infinity)

auto : {uint8}

0 if interval is fixed (default), 1 if the interval can be scaled automatically

name ()

auto

max

min

10.15 timer

Currently, itom does not support the threading module from python. Therefore, no timed calls of python functions are possible using this module. However, the `itom` provides the *timer* class to allow such calls:

10.15.1 timer-Class

class `itom.timer` (`interval, callbackFunc[, argTuple, singleShot]`) → new callback timer

Creates a timer object that continuously calls a python callback function or method with a certain interval. The timer is active after construction and stops when this instance is destroyed or `stop()` is called.

Parameters `interval` : {int}

time out interval in ms

callbackFunc: {function, method} :

Python function that should be called when timer event raises

argTuple: {tuple}, **optional** :

tuple of parameters passed as arguments to the callback function

singleShot: {bool}, **optional** :

defines if this timer only fires one time after its start (True) or continuously (False, default)

isActive () → returns timer status

Returns `status` : {bool}

True if the timer is running, otherwise False.

setInterval (*interval*) → sets timer interval in [ms]

This method sets the timeout interval in milliseconds. The timer calls the callback function continuously after this interval (if started)

Parameters **interval** : {int}

timeout interval in milliseconds. The callback function is continuously called after this timeout once the timer is started.

start () → starts timer

Starts or restarts the timer with its timeout interval. If the timer is already running, it will be stopped and restarted.

stop () → stops timer

10.16 font

class `itom.font` (*family* [, *pointSize* = 0, *weight* = -1, *italic* = false) → creates a font object.

Bases: `object`

This class is a wrapper for the class `QFont` of Qt. It provides possibilities for creating a font type.

Parameters **family** : {str}

The family name may optionally also include a foundry name, e.g. 'Helvetica [Cronyx]'. If the family is available from more than one foundry and the foundry isn't specified, an arbitrary foundry is chosen. If the family isn't available a family will be set using a best-matching algorithm.

pointSize : {int}, optional

If `pointSize` is zero or negative, the point size of the font is set to a system-dependent default value. Generally, this is 12 points, except on Symbian where it is 7 points.

weight : {int}, optional

Weighting scale from 0 to 99, e.g. `font.Light`, `font.Normal` (default), `font.DemiBold`, `font.Bold`, `font.Black`

italic : {bool}, optional

defines if font is italic or not (default)

For an alphabetic index see:

- [genindex](#)

If you need help about other important **Python** modules, like **Numpy**, **Scipy**, **Matplotlib**, see the corresponding references in the internet.

MISCELLANEOUS

11.1 How to use the help

In general you can use this help as any other help.

A problem with function references is, it is hard to keep up to date with the manual. In addition the help can never have all informations about any possible filter- or hardware-plugin delivered from a third party. Therefore some tools for online help were implemented into python and itom.

With `help(method)` you will get an online help for the method or module. To get something similar for the plugin-system, the functions *filterHelp(...)*, *widgetHelp(...)* and *pluginHelp(...)* can be used. If you already have a plugin of type *actuator* or *dataIO* you can get an online-help for possible *parameters* via the member method *getParamListInfo()* and for the exec-functions use *getExecFuncInfo()*.

11.1.1 Rebuild the Help

If you think your help is not up to date and you are using the itom development environment you can rebuilt your help. Therefore you need the up-to-date-version of sphinx for python.

11.2 Units and Conventions

The Units and PlugIn-Parameters are defined according to the SI-Units. Parameters have to be expressed in their unified base unit. (E.g. 1 micron for an actuator setPos-command means 0.001 [mm])

11.2.1 General

- Indexing starts with 0
- The names of PlugIns starts with a capital letter

11.2.2 Physical Units

- Physical Base Units have to be in [mm; s; kg]
- Electrical Base Units have to be in [V; A; W]

DEMO SCRIPTS

There are several python demo scripts available which demonstrate the use of **itom**. All these files are in the directory **demo**. The following list gives a short description of each demo.

12.1 itom Basics

- **demoDataObject.py**

Description: Here you can learn the basic function of the dataObject.

Keywords: creating a dataObject; plot; shallow copy; deep copy; meta data

- **demoToolBar.py**

Description: Creating your own toolbar and buttons.

Keywords: create a new class; add new functions; add button

12.2 Plugins

- **demoDummyGrabber.py**

Description: Usage of a camera plugin.

Keywords: dataIO; start device (camera); snapshot (getVal); live image

- **demoDummyMotor.py**

Description: Usage of a motor plugin.

Keywords: set position; get position

- **demoCMU1394.py**

Description: Firewire grabber for different cameras. PointGray Firefly, Sony SX 900, Sony XCD-X700...

Keywords: dataIO; start device (camera); Snapshot (getVal); Live image

12.3 Algorithm / Filter

- **demoOpenCVFilter.py**

Description: Median filtering of a randomly filled image.

- **demoNumpy.py**

Description: Short demonstration of some linear algebra functions provided by Numpy (numeric package of python).

- **demoScipy.py**

Description: Using scipy and matplotlib to calculate the cross-correlation between two images. Scipy is a python package that contains more scientific algorithms.

Keywords: scipy, matplotlib

- **demoSignalSmooth.py**

Description: Further example on how to use matplotlib (plotting package of python) in itom.

12.4 ui

The subfolder “ui” contains some examples on how to create customized user interfaces in itom (see [Creating advanced dialogs and windows](#)). E.g.:

- **uiMeasureToolMain.py**

Description: Advanced GUI which enables geometric plotting and measurements within a 2D-QWT-Plot. This file shows how to auto-connect to signals and how to use buttons. The corresponding ui-file is uiMeasureToolMain.ui.

PYTHON TUTORIALS

13.1 determine lateral image shift and show images by using itom figure plots

A lateral image shift can be determined by cross correlation. To minimize the calculation time, the inverse fourier transformed product of the fourier transformed lateral shifted images, whereas one image is complex conjugated, is calculated

```
#-----  
#  
# determine the amount of a lateral image shift  
#  
#-----  
  
# import necessary modules  
#  
import numpy as np  
import scipy.misc  
  
#--- create test data (lateral shift an image of Lena) ----  
#  
  
# load an image of Lena  
#  
imageLena = scipy.misc.lena()  
  
# plot lena  
#  
plot(np.flipud(imageLena), 'itom2DQwtFigure')  
  
# amount of pixel shift in x- and y-direction  
#  
xPixelShift = 16  
yPixelShift = -7  
  
# determine the ROI size: relative (centered) size of original image (relativeSize=1: original si  
#  
row, col      = imageLena.shape  
relativeSize = np.floor( min( 1-abs(xPixelShift)/col, 1-abs(yPixelShift)/row ) * 10 ) /10  
  
x0 = int( (col - col*relativeSize)/2 )  
x1 = col-x0 + 1  
y0 = int( (row - row*relativeSize)/2 )  
y1 = row-y0 + 1
```

```
# not shifted ROI
image1 = imageLena[y0:y1,x0:x1].copy()
plot(np.flipud(image1), 'itom2DQwtFigure')

# shifted ROI
image2 = imageLena[y0+yPixelShift:y1+yPixelShift,x0+xPixelShift:x1+xPixelShift].copy()
plot(np.flipud(image2), 'itom2DQwtFigure')

#
#-----

#--- determine the pixel shift -----
#

# discrete fast fourier transformation and complex conjugation of image 2
#
image1FFT = np.fft.fft2(image1)
image2FFT = np.conjugate( np.fft.fft2(image2) )

# inverse fourier transformation of product -> equal to cross correlation
#
imageCCor = np.real( np.fft.ifft2( (image1FFT*image2FFT) ) )

# Shift the zero-frequency component to the center of the spectrum
#
imageCCorShift = np.fft.fftshift(imageCCor)
plot(imageCCorShift, 'itom2DQwtFigure')

# determine the distance of the maximum from the center
#
row, col = image1.shape

yShift, xShift = np.unravel_index( np.argmax(imageCCorShift), (row,col) )

yShift -= int(row/2)
xShift -= int(col/2)

print("shift of image1 in x-direction [pixel]: " + str(xShift))
print("shift of image1 in y-direction [pixel]: " + str(yShift))

#
#-----
```

First the necessary modules have to be imported:

```
# import necessary modules
#
import numpy as np
import scipy.misc
```

Test data is created by lateral shifting a region of interest (ROI) of the Lena image. First the image of Lena is loaded:

```
# load an image of Lena
#
imageLena = scipy.misc.lena()
```

The image of Lena is plotted by using the itom figure plot 'itom2DQwtFigure', which is optimized for 2D static

images. Since the row index for images starts at the top of an image and not at the bottom like for matrixes, the image as to be flipped up side down before plotting.

```
# plot lena
#
plot(np.flipud(imageLena),'itom2DQwtFigure')
```

The ROI size is determined by the amount of lateral shift in x- and y-direction. One ROI is selected from the center of the original image. Another ROI with the same size is shifted from the center about the defined amount.

```
# amount of pixel shift in x- and y-direction
#
xPixelShift = 16
yPixelShift = -7

# determine the ROI size: relative (centered) size of original image (relativeSize=1: original size)
#
row, col = imageLena.shape
relativeSize = np.floor( min( 1-abs(xPixelShift)/col, 1-abs(yPixelShift)/row ) * 10 ) /10

x0 = int( (col - col*relativeSize)/2 )
x1 = col-x0 + 1
y0 = int( (row - row*relativeSize)/2 )
y1 = row-y0 + 1

# not shifted ROI
image1 = imageLena[y0:y1,x0:x1].copy()
plot(np.flipud(image1),'itom2DQwtFigure')

# shifted ROI
image2 = imageLena[y0+yPixelShift:y1+yPixelShift,x0+xPixelShift:x1+xPixelShift].copy()
plot(np.flipud(image2),'itom2DQwtFigure')
```

Now the lateral shift is determined by calculating the inverse fourier transformed of the product of the fourier transformed ROIs and evaluating the distance from the center of the position of its maximum.

```
# discrete fast fourier transformation and complex conjugation of image 2
#
image1FFT = np.fft.fft2(image1)
image2FFT = np.conjugate( np.fft.fft2(image2) )

# inverse fourier transformation of product -> equal to cross correlation
#
imageCCor = np.real( np.fft.ifft2( (image1FFT*image2FFT) ) )

# Shift the zero-frequency component to the center of the spectrum
#
imageCCorShift = np.fft.fftshift(imageCCor)
plot(imageCCorShift,'itom2DQwtFigure')

# determine the distance of the maximum from the center
#
row, col = image1.shape

yShift, xShift = np.unravel_index( np.argmax(imageCCorShift), (row,col) )

yShift -= int(row/2)
xShift -= int(col/2)

print("shift of image1 in x-direction [pixel]: " + str(xShift))
```

```
print("shift of image1 in y-direction [pixel]: " + str(yShift))
```

..toctree::

 maxdepth 1

 todo.rst

INDICES AND TABLES

- `genindex`
- `modindex`

i

itom, [353](#)