SMAP Programming guide

# Introduction to architecture

To ensure modularity as is required for a software project of this complexity, SMAP is extensively using object oriented programming. A large part of the code consists of class-definitions (in +interfaces/) which provide the basic functionality for data handling and user interactions. This is accompanied with several modules providing the overall graphical user interface (GUI) (in +gui/) and a few specific modules for handling localization data (e.g. filtering, defining render parameters, defining ROIs etc). Several small helper functions are found in shared/, and plugins/shared/, as well as their subdirectories, and in private/ directories.

Most functionality, however, is provided by independent modules (in plugins), which define their own user interface and functionality and communicate with SMAP with a global parameter structure and a common localization data object. This communication takes place via a few powerful high-level commands. In SMAP, the modules are loosely sorted into two categories: Workflow modules, that are chained together and can pass on small portions of data (e.g. used for single-molecule fitting) and Dialog modules (BAD NAME; RENAME?), that usually are called once on the whole localization data object to perform e.g. data analysis. However, Dialog modules can be used in workflows, can alter the localization data object and can communicate with other modules, thus the distinction is not very rigid.

In settings/ the user defined GUI appearance, as well as definitions for analysis pipelines are stored.

SMAP supports the rendering and analysis of several ‘layers’, which can be different reconstruction modes, different data sets or channels or even images. This is reflected in many parts of SMAP, where explicitly values are stored per layer.

For own extension of SMAP, please have a look at Documentation/manual/Plugin\_Template.m and Workflow\_Template.m, as well as the function definitions in the respective classes (listed below).

# Back-end classes and GUI

For an own extension of SMAP, it is sufficient to know the main commands to interact with SMAP, thus a detailed understanding is not necessary.

# Global parameter structure

The global parameter structure allows modules to talk to each other, to synchronize fields in their GUI and to notify each other to perform certain actions.

A common par=interfaces.ParameterData object is shared between all modules (obj.P=par), which has to be attached with obj.attachPar(par) upon instantiation of a class.

#### Main methods:

**obj.setPar(name,value)**

name: unique string (char)

value: any data

With this command, you define a parameter name, that can be accessed from any module. If other modules attached a callback function, this is called, if they attached a GUI field, this field is updated.

**Defined in:** interfaces.GUIParameterInterface, interfaces.ParameterInterface

**value=obj.getPar(name)**

name: unique string (char)

value: stored data

Allows access to the stored common paramters.

**Defined in:** interfaces.GUIParameterInterface, interfaces.ParameterInterface

**obj.addSynchronization(name,handle,syncmode,callbackfunction)**

name: unique string (char)

handle: handle to GUI field that is associated to the name and is synchronized with it (can be []).

syncmode: Defines, which pars of the GUI field are synchronized. Cell of strings {‘String’, ‘Value’, ‘AnyOtherProperty’};

callbackfunction: function handle to function which is called when the parameter is changed. This is similar to events and listeners.

Instead of calling obj.addSynchronization you can define the synchroniation in obj.guidef:

pard.syncParameters={{name,handle,synchmode,{@changecallback,obj,parameter}},...};

**Defined in:** interfaces.GUIParameterInterface

#### GUI parameters

When interacting with GUI fields, the values are parsed for simple further evaluation. Numbers (also vectors defined in MATLAB notation) are converted to double, non-parseable numbers are retained as char. For lists, value is a structure with the fields String (cell array with all entries), selection (string of selected entry) and Value (number corresponding to the position in the list of the selected entry).

#### Additional methods:

par=getAllParameters

par=getGuiParameters

par=getSingleGuiParameter(name)

setSingleGuiParameter(name,par)

# Localization data object

The localization data object contains all information about the localizations.

A common locData=interfaces.LocalizationData is common to all SMAP modules (obj.locData=locData) and has to be attached during instantiation of the module with obj.attachLocData(locData).

It has the following fields (not important for programming when using high-level methods):

**.loc:** a structure that stores all data for the single molecule localizations. Every field in the structure is a vector with a length of the total number of localizations. Field names can be arbitrarily defined, but a selection of common field names is given in the appendix.

**.grouploc:** same as loc, but with grouped localization fields. Here, localizations appearing in consecutive frames (dark times of dT allowed) in close vicinity (defined by dX) are combined into one single localization. The length of the vectors corresponds to the number of grouped localizations. These are stored out of performance reasons and can be re-calculated with obj.locData.regroup(dX,dT).

**.layer(layernumber):** structure with the fields .filter and .groupfilter. These are logical values that indicate which localizations are used in the layer defined by layernumber. If a field is missing, it is not used for filtering. These filters are modified by the gui.GuiFilterTable module. If values of the .loc are changed, use locData.filter to recalculate the filters. The final used localizaitons are determined from a logical & over all of these fields.

**.files:** structure with information about the ‘\*\_sml.mat’ files that were loaded. Include microscope metadata, file names and paths and image files associated to a specific file.

**.SE:** Here the ROI manager module is stored (historically called SiteExplorer, thus the name).

Modules can of course directly access the fields of the localizationData object. However, we recommend using the following high-level methods:

**locs=obj.locData.getloc(fields,Name,Value)**

This is the main method to obtain a list of localizations for further use. This powerful command is flexible in returning just a subset of all localizations as specified with the Name,Value pairs.

locs: a structure containing the subset of localizations

fields: an array of localization data fields. It can be any field of locData.loc, in addition: 'ingrouped','inungrouped', which returns the indices of the returned localizations in the .loc and .grouploc vectors.

**Name, Value pairs:**

'layer', layers: double number or vector of layers.

'grouping', string: 'grouped', 'ungrouped' (default), 'layer'(default if ‘layer’ is used). This defines if grouped or ungrouped localizations are returned. If not set and combined with 'layer’, use individual layer grouping. If any of layers is ungrouped, it returns ungrouped localizations.

'channels', channels: double vector of channels

'filenumber', filenumber: double vector of filenumbers

'position', position: 'all' (default), 'roi' (uses defined ROI, if no ROI is defined, it uses the the current FoV), 'fov' (uses the current FoV), double vector: [centerx, centery, widhtx widthy] or [centerx,centery,radius] for user defined rectangular or circular ROI. Define the spatial subset of localizations. If a linear ROI is used, you can request additional fields {‘xnmline’,’ynmline’}, which are the rotated coordinates in the coordinate system of the linear ROI.

'removeFilter', filters: cell array of filter names to remove.

'within', indices which localizations to consider, allows you to simply pass on your own filter.

'shiftxy', shifts=[shiftx, shifty shiftz] by specified vector.

**image=obj.locData.getimage(layer)**

Returns the rendered superresolution data. layer (optional) specifies the layer, if omitted, the composite image of all layers is returned.

image is a structure with the fields: .image: (m\*n\*3) final image, .composite: as .image but without scale and color bars. .rangex, .rangey: coordinates of FoV. .handle: handle to image.

**obj.locData.setloc(field,values)**

Adds a field to the localization data defined by values. The length of values needs to equal the number of localizations. If the field is present, it is overwritten.

**obj.locData.addloc(field,values)**

As obj.locData.setloc, but adds the values below the values present in field. Allows one to add localizations to a data set. If field is not present, it is filled with zeros. This function needs to be called for all fields to avoid fields of different lengths.

#### Additional methods:

.removelocs(indices,removepart)

.saveloc=savelocs(filename,goodind,additionalsave,grouping)

.regroup(dx,dt)

.filter(fields,layers,filtermode,minmax)

# Modules

Most functionality of SMAP is provided by independent modules or plugins. They can be placed in the respective sub-directories of the plugin/ directory and are automatically recognized with a new start of SMAP. The modules provide their own GUI and can be added to quick-access tabs in the SMAP-GUI or called from the plugins menu. They all share the common parameter object and localization data object and talk to each other via the global parameter structure.

## GUI definition

The GUI can be easily and quickly defined by defining a structure in the method guipar=obj.guidef. It is then automatically rendered and formatted, and synchronizations of fields are added.

guipar=obj.guidef: This user-defined method returns a structure. Every field of the structure corresponds to one GUI field. We use the standard notation of uicontrol objects. In addition, you need to set relative positions in .position=[x y] in a 11x4 grid coordinate system. Optionally you can define .Width and .Height in this relative coordinate system. If .Optional is true, this parameter can be rendered with ‘Visible’=’off’ when a simple GUI appearance is chosen.

***Additional fields:***

.plugininfo.name='Plugin Name': name of the plugin

.plugininfo.description='write a description for your plugin';

.plugininfo.type='ProcessorPlugin'; type of plugin. Currently: ProcessorPlugin, WorkflowModule, WorkflowFitter, Renderer, LoaderPlugin, SaverPlugin, ROI\_Analyze, ROI\_Evaluate, WorkflowIntensity. Not used a lot, can usually be omitted.

.inputParameters={'name’}: defines input parameters that are automatically added to p when calling .run(p);

.outputParameters={'guiobject'}; Field of uicontrol defined above which is synchronized to the global parameter structure.

.syncParameters={{'globalParameterName','guiobject2',{'String','Value'},{@aftersync\_callback,obj}}}: define which uicontrols are synchronized with each other. See above.

obj.handle=handle is an important property which defines the position where the GUI is made (handle is the handle to the Parent object of the GUI). Usually it is passed to the constructor of the method, but it can also be set manually before calling obj.makeGUI.

obj.makeGui: Makes the GUI. Part of the class definition. Only call form outaisw, don’t modify.

obj.initGui: Is called after obj.makeGui and allows implementation of additional uicontrols or more elaborate GUIs.

obj.status(text): Writes text into the status line of the main GUI.

## Dialog modules

Dialog modules are modules that are usually called from the Anylyze or Proces tab or the Plugin menu and then perform specific actions on the localization data after pressing the ‘Run’ button (See also the file: Workflow\_Template.m in the Documentation directory).

Methods:

obj.run(p): This method contains the specific code doing the actions. p is a parameter structure containing all GUI parameters as well as the global parameters defined in guidef.inputParameters.

ax=obj.initaxis(name): creates a new tab (or overwrites tab if tab is already present) in results window. Use this for outputs.

Properties:

.history=true/false: when executing run, the GUI parameters of this module are added to the history

.showresults=true/false: opens the results window

To enable Undo for methods that modify the localization data, use in the beginning of the obj.run method the following lines:

obj.setPar('undoModule','PluginName');

notify(obj.P,'backup4undo');

## Workflow modules

Workflow modules are very similar to Dialog modules, but they are meant to be chained together and pass on chunks of data during the analysis. They can be added to and linked in a Workflow object. Additional methods here pass on the data in a synchronized way between modules.

Data is passed on as a interfaces.WorkflowData object with the fields .data (the real data), .frame (the corresponding frame, can be used for synchronization) .ID (additional field that can be used for synchronization), .eof= true only if this is the last data block in the data set.

obj.setInputChannels(inputChannel,syncmode): syncmode = ‘ID’ or ‘frame’ defines which field is used for synchronization.

output=obj.run(data,p). As for the Dialog module, but additionally, the data is passed on. To output data you can either define output=interfaces.WorkflowData (otherwise define as =[]), or specifically call obj.outputdata(data.channel).

obj.prerun(data,p): is called before calling obj.run. This allows lengthy initialization procedures to be called only once.

obj.output(data,outputchannel) outputs the data to the specific channel.

# ROIManager

# Appendix 1: list of important parameter names

# Appendix2: list of important localization data field names