SMAP programming guide

Jonas Ries, EMBL Heidelberg (www.rieslab.de, ries@embl.de)

1 Overview of architecture

SMAP is developed in MATLAB using object-oriented programming. Its functionality is broken down into many modules. Modules are implemented by extending basic classes and provide specific functionality, but also information about the required input parameters to allow a specific GUI to be displayed for each plugin.

SMAP is started by running `SMAP.m`. A SMAP object (in this case just called `g`) is constructed as an object of the class `gui.mainSMAP`. Upon this call, the main GUI is made and many plugin objects are instantiated and their GUIs are displayed within the main GUI.

In addition to superclasses and classes defining specific plugins, SMAP contains many helper functions (in the directory `SMAP/shared` and `SMAP/plugins/shared`).

2 Data Format

Most analyses in SMAP are coordinate-based, i.e. they act on the fitted localizations. Thus, the main data are single-molecule coordinates and other attributes. Combined with additional information, these form a `locData` object (class: `interfaces.LocalizationData`) that is shared between all plugins and can be accessed with `obj.locData`.

2.1 Localization data object

The main properties of the `locData` object are as follows:

`.loc`: This is a structure. Each field of this structure corresponds to one attribute of the localizations (e.g. *x* or *y* coordinate, localization precisions, number of photons, frame, likelihood,…) and is represented as a vector. All these vectors have the same length

(=number of localizations). In case several files are loaded, localizations are just appended to the vectors, the field `filenumber` denotes to which file the localizations belong.

`.grouploc`: Same as above, but containing merged (grouped) localizations. These are calculated when loading a localization data file or when `locData.regroup` is called and stored separately to be immediately available all the time, as the task of grouping can take several seconds.

`.layer`: This property contains all information corresponding to individual reconstructed layers. Specifically, the sub-property `filter` contains logical vectors for each localization attribute/field used for filtering, with `true` meaning that the corresponding localization is used. `groupfilter` contains the respective information for grouped localizations. `images` contains the reconstructed images for each layer and associated information.

`.files`: This contains the vector `file`, here every component corresponds to one loaded file and stores all file-related information. Important fields contain:

`.info`: Metadata associated with the raw image data file

`.tif`: here all added tiff files are stored

`.raw`: a subset of the raw camera frames are stored

`.savefit`: a structure with the fitting parameters

At this location also other results of plugins that directly change the localization data is stored, such as `.driftinfo`, `.transform`, `.transformation` etc.

`.history`: A cell array with each component containing the parameters of a specific plugin that was applied to the localization data.

2.2 Accessing localization data

`locs=obj.locData.getloc(fields,parameter1,value1,...)` is a high-level method of the localization data class to access localization attributes for a defined subset of localizations. `locs` is a structure with fields corresponding to attributes of the localizations. `fields` is a cell array of the fields that you want to read out. Additional parameter/value pairs describe precisely what subset of localizations you want to access.

You can use this method to access filtered or unfiltered localizations, those only in a defined ROI or those displayed in the main reconstruction window. You can access localizations in specific layers, and define if to read out grouped or ungrouped localizations. This method makes it easy to define in the main GUI what part of the localizations to use for a specific analysis plugin.

For example, to return *x* and *y* coordinates that are displayed in layer 1 and layer 2 inside a user-defined ROI use:

```
locs=obj.locData.getloc({'xnm', 'ynm'},'layer',[1 2], 'position',...
    'roi')
```

See `SMAP/+interfaces/private/getlocs.m` for a full description of parameter/value pairs.

2.3 Adding attributes to localization data

`obj.locData.setloc(name,value)` adds the attribute `name` to the localization data. `value` needs to be a vector with a length corresponding to the number of localizations. With `obj.locData.regroup` you can update the grouped localizations.

3 Parameter sharing and synchronization

All SMAP objects contain the same shared `locData` object as a property. However, this object is only used to share data-specific parameters. In addition, all components can

communicate via a parameter object. This is another shared object, stored as an object property `obj.P`.

3.1 Accessing parameters

Parameters are accessed via a user-defined `name`. Their `value` can be any MATLAB data format or object, but large structures or objects lead to loss in performance. The main functions are defined in the `GuiParameter` superclass (subclassed by all SMAP plugins) and accessible as:

`value=obj.getPar(name)`: Returns the value associated with `name`.
`obj.setPar(name, value)`: Sets the parameter `name` to the specified `value`.

3.2 Synchronization

Parameters can be synchronized with GUI controls. Whenever a user changes a control, this changes the parameter value and changes the control of any other synchronized GUI component in other plugins to the same value. This provides a simple way of synchronizing GUI parameters between different plugins. In addition, a function can be defined that is called after a parameter is changed with `obj.setPar` or in the GUI, allowing any plugin to react to changes in parameters (this is similar to the `notify` concept in MATLAB).

You can add a synchronization with:

`obj.addSynchronization(ParameterName,handle,syncmode,changecallback)`, as defined in `interfaces.GuiParameterInterface.m`.

Here, the function arguments have the following meaning and can take the following values:

`ParameterName`: shared parameter name. This creates the shared parameter field.
`handle`: handle to GUI control (guiobject) to be synchronized. Typically the guiobject is accessible as a property of 'obj.guihandles'.
`syncmode`: what to synchronize. Use either 'String', 'Value', or 'other property', depending on the nature of the parameter.
`changecallback`: function handle to function which is called when parameter is changed. This is similar to events and listeners.

Alternatively, and probably simpler, add it to the definition of the plugin GUI (`function p=obj.guidef`) with:

```
p.syncParameters={{'ParameterName','guiobject',syncmode,@obj.aftersync_callback}};
```

`ParameterName`: Name of the shared parameter
`guiobject`: Name of the GUI object to be synchronized
`syncmode`: Cell array of what properties to synchronize (e.g. `{'String', 'Value'}`)
`@obj.changecallback`: Function handle to the function that is called when the parameter is changed.
See `SMAP/Documentation/Manual/Plugin_Template.m` for further explanations.

3.3 GUI parameters

All GUI controls (see section 4.2 on how to create those) are linked to parameters. Handles to GUI controls are stored in `obj.guihandles`. With `p=obj.getAllParameters` these are converted into a structure. This `p` is also directly passed on to the `run(obj,p)` method of a plugin. Editable controls that contain values (or vectors) are automatically converted to numerical variables, checkboxes are converted to logicals and popup menus return a structure with `string` (all entries), `value` (number of selected entry) and `selection` (string of selected entry).

Get a single GUI parameter with `obj.getSingleGuiParameter('name')`.

Set GUI parameters with `obj.setGuiParameters(p)`. p is a structure similar to that obtained with `obj.getAllParameters`. Every field of p should have the same name as a GUI control element. Its value is converted back to a value/state of the specific control.

4 Overview of plugin structure

All analysis plugins are subclasses of the class `interfaces.DialogProcessor`. For implementing own plugins, we recommend starting with an existing plugin or preferably with the template `SMAP/Documentation/Manual/Plugin_Template`. New plugins that are saved in a sub-directory of `SMAP/plugins` are automatically recognized upon starting SMAP and added to the menu.

In the class constructor you can set several switches for the plugin behavior (see template for details). The GUI is automatically generated based on information in `obj.guidef` (see section 4.2 for details). All handles to GUI components are stored in `obj.guihandles`. After the GUI is created the method `obj.initGui` is called, here you can add functionality to change the GUI or initialize the plugin.

All analysis plugins contain a Run button to start the analysis. This calls the method `obj.run(p)`. All GUI parameters are passed on in the structure `p`.

Usually, localizations to be analyzed are obtained with `locs=obj.locData.getloc(…)` (see section 2.2).

To avoid opening many figure windows as outputs of plugins, there is a default output window. This output window is created invisible if `obj.showresults=false` (default), but is made visible if ☐show results is checked in the GUI. Every output creates a new tab. Create a new tab with `ax=obj.initaxis('name')` and access it via the axis handle `ax`.

4.1 Plugin GUI

SMAP plugins have a simple syntax to define a GUI. However, you can also manually create a GUI by either overwriting the `obj.makeGui` method or by using `obj.initGui`. Don't forget to store handles to new GUI controls as fields in `obj.guihandles`.

To make use of the automatic GUI creation, change the `pard` structure in `pard=obj.guihandles` (see `SMAP/Documentation/Manual/Plugin_Template` for details). In short, every control corresponds to a field in `pard`, the name of the field is the common identifier (this will also be the name of the associated parameter). The value of the field is again a structure with the following fields:

`object`: structure defining the GUI control. These correspond to name, value pairs of the MATLAB `uicontrol` function. In case you want to pass on a cell (e.g. string of a `popupmenu`) you need to use double brackets `{{'str1', 'str2'}}`.

`position`: Position of the GUI control on a 7 x 4 grid in the GUI panel.

`Width`: Width of the GUI control in units of the grid (optional, default =1).

`Height`: Height of the GUI control in units of the grid (optional, default =1).

`TooltipString`: Tool tip associated with the control; this is shown when the mouse hovers over the control (optional).

`Optional`: If this is `true`, this control is hidden from view if the simple GUI is selected (optional, default=`false`).

You can hide and show GUI controls dependent on the state of a specific control and define the name and description for the plugin, as well define a synchronization of GUI controls (for details see `SMAP/Documentation/Manual/Plugin_Template.m` and section 3.2).

4.2 Programmatically accessing SMAP plugins

You can create an object instantiating a plugin at the path path1/path2/pluginname with `o=plugins('path1','path2','pluginname')`, e.g. with `o=plugin('Analyze','cluster','DBSCAN_cluster')`. Attach the `locData` object (e.g. attached to the SMAP object `g.locData`, or in any other plugin as `obj.locData`) with `o.attachLocData(locData)`. Attach the parameter structure `P` (from `g.P` or `obj.P`) with `o.attachPar(P)`. If you want to create a GUI, set `o.handle=figurehandle` to the figure in which you want the GUI to be positioned, set `o.guiPar.Vrim=100` to position the GUI in the parent figure, and make the GUI with `o.makeGUI`. You can either set all parameters in the structure `p` and execute the plugin with `o.run(p)`, or you write all parameters in the GUI with `o.setPar` (see section 3.1) and call `results=o.processgo`. This calls `o.run(p)`, makes the results window, returns the results and, if set in the object properties, backs up the localization data for the 'undo' function and saves the 'history'.

If you want to interact with plugins that were already instantiated by the main SMAP GUI you can access them at `o=g.children.category.children.plugin`. Sometimes you have to go deeper into the structure, e.g. for the `guiSites` category.

5 Workflows

5.1 Workflow architecture

Workflows are a collection of plugins that are chained and act sequentially on data. For every plugin the next plugin is defined and processed data is passed on. Data is passed on as part of an `interfaces.WorkflowData` object. Workflow plugins (see section 4.3) are special plugins that can receive data, processes them and passes them on to the next plugin. Standard plugins on the other hand act directly on the `LocalizationData` and can also be added in workflows.

As workflow plugins can have more than one input channel (i.e. they can receive data from several previous workflow plugins), data synchronization becomes important, so that always the corresponding data (e.g. for SMLM fitting plugins the data corresponding to the same frame) are processed at the same time.

By default, this synchronization is controlled via the 'frame' property of the WorkflowData object. Then only if all input channels have a WorkflowData object with the same 'frame' property (i.e. all previous plugins have already processed this frame), the plugin is called (`datout=obj.run(datin,p)`) and the data object is removed from the buffers. Instead of 'frame' also other properties can be used for synchronization. The synchronization is controlled by calling (in the constructor) the method:

```
obj.setInputChannels(numberOfInputChannels,syncmode);
```

`syncmode` can be `'frame'` (default), `'ID'` (you can define any other parameter to synchronize and put it in the ID field of the WorkflowData object) or `'none'` (no synchronization carried out, the `run` function is called as soon as the first channel is available.

The buffers that pass on data between plugins are meant to carry all large data. Other parameters, especially those that do not change dynamically, are usually shared via the obj.setPar and obj.getPar functions (see section 3.1).

5.2 Assembling workflows

The workflow itself is assembled using its GUI. Make a new workflow using SMAP **Menu/Plugins/New workflow.** Add workflow plugins to the workflow by right-clicking on the plugin list. For each plugin you need to select which other plugin passes on the data to this plugin. Save will save the workflow object, Graph will display a graphical representation of the workflow.

To use a workflow as a fitting workflow in SMAP, you can load the workflow in the **Localize** tab by right-clicking on the tabs and selecting add workflow in the context menu. It is however more convenient to load the workflow with the Change button. But for this, the GUI controls of each workflow plugin need to be mapped to different tabs. This is done by a .txt file with the same name as the workflow (see `Workflow_Template.txt`). It needs to contain the following lines (remove %comments):

`all.file=relative/path/nameOfWorkflow.mat` %where to find the workflow file

`all.FieldHeight=25` %height of fields in pixels

Define all tabs with

`tab.tabname.name=Name of Tab`

and all plugins (with the name `PluginX`) with

`PluginX.handle=tabname` %in which tab to position the GUI

`PluginX.Vpos=1` % where to position it vertically (Nx4 grid)

`PluginX.Xpos=2` % where to position it horizontally

5.3 Workflow plugins

To generate your own workflow plugins we recommend starting with an existing plugin or the template `Documentation/Manual/Workflow_Template.m`. Workflow plugins are similar to Analysis plugins (section 4), with the GUI defined in the same way. You need to implement the following methods:

1. If you have more than one input channel then set the number of input channels with `obj.inputChannels=N` in the constructor.
2. The `obj.prerun(p)` method is called once before starting the workflow. You can initialize the workflow here. `p` contains a structure with all GUI parameters.
3. The main functionality is contained in `obj.run(datain,p)`. `datain` is an `interfaces.WorkflowData` object, see `Workflow_Template.m` for additional information. Data is stored in `datain.data1`.
4. The last `WorkflowData` object to be processed (last data block) has `datain.eof=true`.
5. A plugin uses the `data1` passed to generate an output data `data2`.
6. This needs to be written in an `interfaces.Workflow` object. You can copy `datain` to `dataout` and replace the data part with `dataout.data=data2`;
7. Triggering the output generation either directly using `obj.output(dataout)` or by returning the variable `output=dataout` from the `obj.run` function then will pass the data object to the next plugin.

6 ROI Manager

6.1 The ROI manager object

The ROI manager stores the positions and annotations of specific ROIs (also called sites) that may originate from many files. It is an object `se=interfaces.SiteExplorer` that is attached to `obj.locData.SE=se` and thus shared with all plugins. Its properties contain three vectors of `interfaces.SEsites` that are lists with `files`, `cells` and `sites` (ROIs). The main properties of `interfaces.SEsites` are (not all of them are used for the `cells` and `files` properties):

`pos`: absolute position of the ROI center in nanometers
`ID`: ID of the ROI
`info`: structure which cell and file the ROI belongs to
`annotation`: structure with annotations defined in the ROI GUI
`evaluation`: structure with results from evaluation plugins
`name`: name of the ROI
`image`: rendered superresolution image of the ROI.

6.2 Evaluation Plugins

Evaluation plugins are plugins that evaluate single ROIs, specifically single objects of the class `interfaces.SEsites`. Usually, they are run on all ROIs of a data set and output results of the analysis.

They are subclasses of `interfaces.SEEvaluationProcessor` and can be programmed in the same way as normal analysis plugins (section 4).

The functionality is encoded in the method `out=obj.run(p)`.

Instead of obtaining localization data with `locs=obj.locData.getloc`, here we recommend using analogous code `locs=obj.getloc`. If no Position parameters are passed on, the position of the ROI is used.

The output of the evaluation is a structure with `out.property=value`. It is stored in the `interfaces.SiteExplorer` object as:

```
SE.sites(k).evaluation.pluginname.property=value
```

6.3 Analysis of ROI manager evaluations

The results of the evaluation plugins are stored at

```
SE.sites(k).evaluation.pluginname.property
```

You can get a vector of properties with the command:

```
property=getFieldAsVector(SE.sites,'evaluation.pluginname.property')
```

In this way you can get all evaluation results and use them for further analysis / statistics. When you write ROI manager analysis plugins, you can use this notation.