

# McStasScript

Mads Bertelsen

June 16, 2021

## 1 Introduction

This document serves as the documentation for the McStasScript scripting language for python. The purpose of McStasScript is to generate McStas/McXtrace instrument files from python which is simply another way of writing an instrument file. The main advantages are the possibility of using for-loops and that it can be used directly from a python terminal. The simulation described by the instrument can be executed from the scripting language and the data can be manipulated before plotting. It is possible to convert existing McStas/McXtrace instruments to a python version using an included converter.

## 2 Installation

The package can be installed/updated through pip with the following terminal command.

```
1 python3 -m pip install McStasScript --upgrade
```

Examples are available in the github repository ([link](#)). The package can also be obtained directly from github, but in this case the python path has to be set manually before importing, for example:

```
1 import sys
2 sys.path.append('/Users/madsbertelsen/PaNOSC/McStasScript') # Path to package
```

## 3 Importing the package

The code is structured as a python package where the classes and functions meant for the user are to be imported. The important classes and functions are contained in the interface package, and are called instr, plotter, functions and reader.

```
1 from mcstasscript.interface import instr, plotter, functions, reader
```

## 4 Configuration

McStasScript needs to know where to find the McStas installation it should use. This information is stored in a configuration file located with the python package, but can be updated through a Configurator class. This configuration is permanent and is only performed for first use or when updating McStas. The default values are for a Mac running McStas version 2.5 and are shown here. The line length is also set for comfortable use in jupyter notebooks.

variable	default
mcstas_path	/Applications/McStas-2.5.app/Contents/Resources/mcstas/2.5/
mcrun_path	/Applications/McStas-2.5.app/Contents/Resources/mcstas/2.5/bin/
mcxtrace_path	/Applications/McXtrace-1.5.app/Contents/Resources/mcxtrace/1.5/
mrxrun_path	/Applications/McXtrace-1.5.app/Contents/Resources/mcxtrace/1.5/bin/
characters_per_line	93

The values are updated using the Configurator class which is loaded from the interface package. This can be done from a python file or a Jupyter Notebook. Here they are updated to values appropriate for an Ubuntu system.

```
1 from mcstasinterface import instr, plotter, functions
2 my_configurator = functions.Configurator()
3 my_configurator.set_mcrun_path("/usr/bin/")
4 my_configurator.set_mcstas_path("/usr/share/mcstas/2.5/")
5 my_configurator.set_mrxrun_path("/usr/bin/")
6 my_configurator.set_mcxtrace_path("/usr/share/mcxtrace/1.5/")
7 my_configurator.set_line_length(120)
```

For a Windows system, paths use backslash instead of forward slash, and as this is the escape character, double backslashes are necessary. A default Windows McStas installation would require these paths.

```
1 my_configurator.set_mcrun_path("\\mcstas-2.6\\bin\\")
2 my_configurator.set_mcstas_path("\\mcstas-2.6\\lib\\")
3 my_configurator.set_mrxrun_path("\\mcxtrace-1.5\\bin\\")
4 my_configurator.set_mcxtrace_path("\\mcxtrace-1.5\\lib\\")
```

## 5 Documentation

This section describes how to use the package by going through the available classes and their methods.

### class McStas\_instr/ McXtrace\_instr

*Holds methods for creating a McStas or McXtrace instrument file*

Initiating an instance of the class requires a name to be given as the first argument, but it is possible to add more information using optional keyword arguments. In this manual McStas\_instr will be used for the examples, but the syntax is identical for the McXtrace\_instr class. The author and origin can be specified, and the package path from the configuration can be overwritten if necessary. In the table below the positional arguments are above the dashed line and the keyword arguments are below.

input	type	explanation
first argument	string	name of the instrument
author	string	name of the author
origin	string	origin of the work
input_path	string	path work directory (default is current work directory)
executable_path	string	path to local binary (overwrites default from config file)
package_path	string	path to package directory (overwrites default from config file)

Below two instrument objects are created, one called *detector* with an instrument named "LOKI\_detector" that has its origin set to "DMSC", and another called *example* with an instrument named "test" with the author specified as "Mads Bertelsen". When instrument files are written, they will have the specified instrument names with the ".instr" extension.

```

1 detector = instr.McStas_instr("LOKI_detector", origin="DMSC")
2 example = instr.McStas_instr("test", author="Mads Bertelsen")

```

When the instrument object is created, components in the current work directory are loaded. It is possible to define a work directory different from the python work directory using the *input\_path* keyword argument. Instrument files are written to the McStasScript work directory, and mcrun is executed within that folder.

## McStas\_instr method add\_parameter

*Adds input parameter to instrument, uses class parameter\_variable*

input	type	explanation
first argument (optional)	string	variable type
second argument	string	name of the parameter
value	any	default value for the parameter
comment	string	comment that will be displayed with the variable

Here four different parameters are added to the instrument file using the different allowed keywords.

```

1 detector.add_parameter("wavelength")
2 detector.add_parameter("double", "height", value=1.0, comment="Height in [m]")
3 detector.add_parameter("string", "reflection_filename", comment="Stored reflections")
4 detector.add_parameter("string", "data_filename", value="\data.dat", comment="Data")

```

The two first variables called *wavelength* and *height* are of the default type because no type was given. In McStas the default type is a double. The *height* variable was given a default value and a comment. The *reflection\_filename* and *data\_filename* are both specified to be strings and the latter was given a default value. Note the `\` needed to insert the quotation marks into strings.

## McStas\_instr method show\_parameters

*Shows currently defined parameters in the instrument*

This method is useful when running the simulation to get an overview of the available instrument parameters.

```

1 detector.show_parameters()
2     wavelength
3 double height = 1.0 // Height in [m]
4 string reflection_filename // Stored reflections
5 string data_filename = "data.dat" // Data

```

## McStas\_instr method add\_declare\_var

*Adds declared variable to the instrument file*

input	type	explanation
first argument	string	variable type
second argument	string	name of the parameter
value	any	value for the parameter (can be array)
array	int	length of array
comment	string	comment that will be displayed with the variable

Here four different variables are added to the instrument file using some of the different allowed keywords.

```

1 detector.add_declare_var("double", "energy")
2 detector.add_declare_var("int", "flag")
3 detector.add_declare_var("double", "tube_radius", value=0.013)
4 detector.add_declare_var("double", "displacements", array=7)
5 detector.add_declare_var("double", "t_array", array=3, value=[0.65E-6, 0.65E-6, 1E-6])

```

When declaring an array the array keyword must be used even when setting the values. The values are given as a python array as shown in the last example. The declared variables will appear in the declare section of the instrument file.

### McStas\_instr method `append_declare`

*Adds line of code to declare section*

Most declarations can be handled using the `add_declare_var` method, but for more advanced use such as adding structures or functions, lines of code can be added directly.

```

1 detector.append_declare("double square(double x)")
2 detector.append_declare("{")
3 detector.append_declare("    return x*x;")
4 detector.append_declare("}")

```

### McStas\_instr method `append_initialize`

*Adds line of code to initialize section*

This methods adds a line of code to the initialize section of the McStas file and has no keyword arguments. A similar method called *append\_initialize\_no\_new\_line* exists for adding to the same line with multiple calls.

```

1 detector.append_initialize("energy=pow(2*PI/wavelength*K2V,2)*VS2E;")

```

### McStas\_instr method `show_components`

*Shows currently available McStas components*

Before adding components to our instrument, it is nice to get an overview of the available components. The method `show_components` can be called without arguments, and will show the available categories of McStas components such as sources, optics and samples.

input	type	explanation
first argument (optional)	string	name of category to show components in

By specifying a category, the components in that category is shown.

```

1 detector.show_components("samples")

```

```

1 Here are all components in the samples category.
2 Incoherent      Phonon_simple  Res_sample      Single_crystal
3 Isotropic_Sqw   Powder1        Sans_spheres    TOFRes_sample
4 Magnon_bcc      PowderN        SasView_model   Tunneling_sample

```

### McStas\_instr method `component_help`

*Shows parameters, their defaults and an explanation for given component*

input	type	explanation
first argument	string	name of component

The text is shown with some additional formatting highlighting which parameters are required and optional, along with what the default values are. This information is loaded directly from the local component file, and any component in the work directory will take priority over the standard version.

```

1 detector.component_help("Phonon_simple")
2
3 --- Help Phonon_simple -----
4 radius [m] // Outer radius of sample in (x,z) plane
5 yheight [m] // Height of sample in y direction
6 sigma_abs [barns] // Absorption cross section at 2200 m/s per atom
7 sigma_inc [barns] // Incoherent scattering cross section per atom
8 a [AA] // fcc Lattice constant
9 b [fm] // Scattering length
10 M [a.u.] // Atomic mass
11 c [meV/AA(-1)] // Velocity of sound
12 DW [1] // Debye-Waller factor
13 T [K] // Temperature
14 target_x = 0 [m] // position of target to focus at . Transverse coordinate
15 target_y = 0 [m] // position of target to focus at. Vertical coordinate
16 target_z = 0 [m] // position of target to focus at. Straight ahead.
17 target_index = 0 [1] // relative index of component to focus at, e.g. next is +1
18 focus_r = 0 [m] // Radius of sphere containing target.
19 focus_xw = 0 [m] // horiz. dimension of a rectangular area
20 focus_yh = 0 [m] // vert. dimension of a rectangular area
21 focus_aw = 0 [deg] // horiz. angular dimension of a rectangular area
22 focus_ah = 0 [deg] // vert. angular dimension of a rectangular area
23 gap = 0 [meV] // Bandgap energy (unphysical)

```

## McStas\_instr method add\_component

*Method for adding a new component to the instrument file*

A McStas component describes a part of the instrument including its position and rotation in space. When adding a new component in McStasScript, the name and type must be specified. The *add\_component* method returns the appropriate component object that can be manipulated directly, but it is also possible to manipulate through methods in McStas\_Instr. Most commonly a component is added to the end of an instrument file, but the keyword arguments *before* or *after* can be used to place the component before/after a previously specified component. All component classes are dynamically generated based on components in your local McStas installation and in the python work directory, and in this way have all input parameters as class attributes.

input	type	explanation
first argument	string	name of the component instance
second argument	string	name of the component to use
AT	float list[3]	position in (x,y,z) [m]
AT_RELATIVE	string	name of earlier component used as reference for position
ROTATED	float list[3]	rotation angles around (x,y,z) [deg]
ROTATED_RELATIVE	string	name of earlier component used as reference for rotation
RELATIVE	string	name of earlier component used as reference
before	string	name of component this component should be before
after	string	name of component this component should be after
WHEN	string	WHEN statement (McStas keyword)
EXTEND	string	EXTEND c code (McStas keyword)
GROUP	string	GROUP name (McStas keyword)
JUMP	string	JUMP string (McStas keyword)
SPLIT	int	SPLIT value (McStas keyword)
c_code_before	str	c code inserted before component (for exmaple %include)
c_code_after	str	c code inserted after component
comment	string	comment that will be displayed with the variable

A component in McStas needs a name, which is the first argument. The second argument select what component should be used from the component library. Below are some examples of simple use.

```

1 detector.add_component("Origin", "Arm")
2 src = detector.add_component("source", "Source_simple", RELATIVE="Origin")
3 detector.add_component("beam_extraction", "Guide_gravity",
4                         AT=[0,0,2], RELATIVE="source")

```

Here *src* would be a python object that can be modified to change the source. If one wishes to insert another component between the source and beam\_extraction it can be done with the *before* or *after* keyword.

```

1 detector.add_component("pre_guide_slit", "Slit", before="beam_extraction",
2                         AT=[0,0,1], RELATIVE="source", comment="Slit before the guide")

```

## McStas\_instr method print\_components

*Method for printing current list of components to the terminal*

To check that the components defined in the documentation so far are in the expected order, the print\_components method is demonstrated. Data on the rotation of components is normally included but is omitted for clarity. This method has no arguments.

```

1 detector.print_components()

```

1	Origin	Arm	AT	[0, 0, 0]	ABSOLUTE
2	source	Source_simple	AT	[0, 0, 0]	RELATIVE Origin
3	pre_guide_slit	Slit	AT	[0, 0, 1]	RELATIVE Origin
4	beam_extraction	Guide_gravity	AT	[0, 0, 2]	RELATIVE source

## McStas\_instr method set\_component\_parameter

*Method for setting parameters of a component using a dictionary*

This method sets the parameters of a defined component using a python dictionary.

input	type	explanation
first argument	string	name of the component instance to modify
second argument	dict	dictionary with parameter names and values

It is possible to add several parameters in one call, and new calls add further parameters.

```
1 detector.set_component_parameter("source", {"xwidth" : 0.12, "E0" : "energy"})
2 detector.set_component_parameter("source", {"yheight" : 0.12})
```

An error will occur if the given parameter name does not match a parameter in the component type.

## McStas\_instr method `print_component`

*Method for printing information contained in defined component*

This method takes the name of a component and prints the current information. We can check that the parameters and position of a component have been registered correctly.

```
1 detector.print_component("source")

1 COMPONENT source = Source_simple
2   yheight = 0.12 [m]
3   xwidth = 0.12 [m]
4   E0 = energy [meV]
5 AT [0, 0, 0] RELATIVE Origin
6 ROTATED [0, 0, 0] RELATIVE Origin
```

Note that this is not intended for copy-pasting into McStas instruments as the syntax is not correct. Generation of the instrument file is covered later in the documentation. The units are collected from the header file of the component definition. If a required parameter has not yet been specified, the user will be reminded when using this method.

## McStas\_instr method `set_component_AT`

*Method for updating position of a component*

There is a range of methods for updating information on a component after it has been defined. The syntax is similar to the original call for *add\_component* in all cases.

input	type	explanation
first argument	string	name of component to modify
first argument	float list[3]	position in (x,y,z)
RELATIVE	string	name of earlier component used as reference for position

```
1 detector.set_component_AT("source", [0.01, 0, 0])
```

## McStas\_instr method `set_component_ROTATED`

*Method for updating rotation of a component*

input	type	explanation
first argument	string	name of component to modify
second argument	float list[3]	rotation angle around (x,y,z) [deg]
RELATIVE	string	name of earlier component used as reference for rotation

The rotation will only be written to the McStas instrument file if specified. When no rotation is present, McStas defaults to no rotation being applied.

```
1 detector.set_component_ROTATED("beam_extraction", [0, 2.0, 0], RELATIVE="Origin")
```

## McStas\_instr method set\_component\_RELATIVE

*Method for updating RELATIVE reference for both position and rotation*

This method will override both positional relative and rotational relative. It has no keyword arguments.

```
1 detector.set_component_RELATIVE("beam_extraction", "pre_guide_slit")
```

After these updates the output from print\_components is shown again to see the changes. The Origin component show the default settings.

1	Origin	Arm	AT	[0, 0, 0]	ABSOLUTE
2	source	Source_simple	AT	[0.01, 0, 0]	RELATIVE Origin
3	pre_guide_slit	Slit	AT	[0, 0, 1]	RELATIVE Origin
4	beam_extraction	Guide_gravity	AT	[0, 0, 2]	RELATIVE pre_guide_slit

1	Origin	Arm	ROTATED	[0, 0, 0]	ABSOLUTE
2	source	Source_simple	ROTATED	[0, 0, 0]	RELATIVE Origin
3	pre_guide_slit	Slit	ROTATED	[0, 0, 0]	RELATIVE Origin
4	beam_extraction	Guide_gravity	ROTATED	[0, 2.0, 0]	RELATIVE pre_guide_slit

## McStas\_instr method set\_component\_WHEN

*Method for setting WHEN condition on component*

The input for this method is a string, which should be a  $C$  logical expression involving variables defined in declare and the state parameters of the neutron.

```
1 detector.set_component_WHEN("beam_extraction", "vx > 0")
```

## McStas\_instr method append\_component\_EXTEND

*Method for adding a line to the extend section of a component*

The EXTEND section adds code to a McStas component and its scope includes variables declared in the instrument file and the component. The number of scattering events in a component can for example be saved to an external parameter using the SCATTERED keyword. Two events are subtracted since entering and leaving the guide counts as a scattering event.

```
1 detector.append_component_EXTEND("beam_extraction", "n_scattering = SCATTERED - 2")
```

## McStas\_instr method set\_component\_GROUP

*Method for setting GROUP name of a component*

The GROUP keyword is used to make a number of components parallel in the execution, however the order still matters. It could for example be used if several guides were simulated after the source, and each of them would be in the same group.

```
1 detector.set_component_GROUP("beam_extraction", "guides")
```

## McStas\_instr method set\_component\_JUMP

*Method for setting JUMP statement of a component*



The JUMP keyword is an advanced feature of McStas that is similar to a goto. The string given to the method should contain what follows the JUMP keyword in a McStas component. An example with the correct syntax is shown below, but there is no point in iterating over a guide in this case.

```
1 detector.set_component_JUMP("beam_extraction", "myself iterate 3")
```

## McStas\_instr method set\_component\_SPLIT

*Method for setting SPLIT value of a component*

The McStas SPLIT keyword will split the ray going into a component evenly into a given number of rays whose total weight is equal to the initial weight. This is useful for example when a complex guide system takes a lot of computation time and the sample has Monte Carlo choices. It is always important that the component after the split has Monte Carlo choices, as the same ray will otherwise just be simulated in an identical manner many times, ultimately achieving the same result with more time spent.

```
1 detector.set_component_SPLIT("powder_sample", 300)
```

## McStas\_instr method set\_component\_c\_code\_before

*Method for setting C code to be printed before component*

This method is most often used for inserting `%include` statements that insert components from another instrument to that point in an instrument. A similar method called `set_component_c_code_after` is available to insert code after the component.

```
1 detector.set_component_c_code_before("powder_sample", "%include ILL_H22.instr")
```

## McStas\_instr method set\_component\_comment

*Method for updating the comment on a component*

Using this method a comment can be set for the specified component.

```
1 detector.set_component_comment("beam_extraction", "Simulating severe misalignment")
2 detector.print_component("beam_extraction")
```

```
1 // Simulating severe misalignment
2 COMPONENT beam_extraction = Guide-gravity
3 AT [0, 0, 2] RELATIVE pre-guide-slit
4 ROTATED [0, 2.0, 0] RELATIVE pre-guide-slit
```

## McStas\_instr method copy\_component

*Copies component instance*

input	type	explanation
first argument	string	name of new component
second argument	string	name of original component
Keyword arguments		All keyword arguments from <i>add_component</i> allowed

This method can be used to copy components already defined. It is possible to use all keyword arguments from the `add_component` method here, which will be applied to the new component.

```
1 detector.copy_component("beam_extraction_thermal", "beam_extraction",
2                        AT=[0.12, 0, 0], AT_RELATIVE="beam_extraction")
```

## McStas\_instr method write\_c\_files

*Methods for writing C files to folder named generated\_includes*

This method will write C files describing the *declare*, *initialize* and *trace* sections of the generated instrument.

```
1 detector.write_c_files()
```

These can then be included in another McStas file. This is useful as this python tool is most often used to generate large repeating part of an instrument that can then be included in a regular instrument file. The instrument file can include them using the *%include* keyword from McStas as shown below.

```
1 DECLARE
2 %{
3 // include parameters declared from generate_LOKI_parts.py
4 %include "generated_includes/LOKI_detector_declare.c"
5 %{
6
7 INITIALIZE
8 %{
9 // include initialization code from generate_LOKI_parts.py
10 %include "generated_includes/LOKI_detector_initialize.c"
11 %{
12
13 TRACE
14 // include components from generate_LOKI_parts.py
15 %include "generated_includes/LOKI_detector_component_trace.c"
```

## McStas\_instr method write\_full\_instrument

*Writes the full instrument file with name defined in original McStas\_instr call*

This method instead writes the entire instrument file using the provided information.

```
1 detector.write_full_instrument()
```

## McStas\_instr method run\_full\_instrument

*Runs McStas simulation of defined instrument and returns array of McStasData objects*

This method runs the simulation using the *mcrun* commands of the system and returns the resulting data as a list of McStasData objects. Normally an error will occur if the *foldldername* already exists. By setting the *increment\_folder\_name* keyword argument to *True*, the *foldldername* is updated automatically for each run to avoid this. It is default behavior that a new instrument file is written at every run, requiring a new compile step, if this is not desired the *force\_compile* keyword argument can be used. Be aware that with *force\_compile* set to *False*, no new instrument is written and the last compiled instrument is used for the run.

input	type	explanation
foldldername	string	name of folder that will be created for data
parameters	dict	Dictionary with input parameters and their values
ncount	int	Number of rays to simulate
mpi	int	Number of mpi threads to use for simulation
custom_flags	string	Custom mctas flags added to mcrun launch command
executable_path	string	Absolute path to mcrun/mxrun (overwrites path from config file)
force_compile	bool	If false, wont write new instrument and will reuse old compile
increment_folder_name	bool	If True, increments data folder name automatically
suppress_output	bool	If True, no text output will be shown

If the `mpi` keyword argument is not set, the `mcrun` command will use the standard *C* compiler and run using a single core. Setting `mpi` to 1 results in `mcrun` using the configured `mpi` enabled *C* compiler, which may require additional steps during the McStas installation. Below an example of a simulation run is shown.

```
1 data = detector.run_full_instrument(foldername="data1",
2                                   parameters= {"wavelength":5.1},
3                                   ncount=1E7, mpi=2)
```

## McStas\_instr method interface

*Provides widget simulation interface in jupyter notebook*

This method will only work correctly in a jupyter notebook, and will show an interface meant for running simulations. It is required to use matplotlib widget mode as shown in the example. An example of the interface can be seen on figure 5. All parameters can be set as text, or if they have options included as dropdown menus. The `ncount` can be set along with number of simulation threads. A button is provided for starting a new simulation. When a simulation run has been performed, the data is shown and can be viewed with the options on the right hand side, these are identical to those in the `plotter.interface`.

```
1 %matplotlib widget
2 detector.interface()
```

## McStas\_instr method show\_instrument

*Shows McStas instrument using mcdisplay*

This method calls the `mcdisplay` command which will display a geometrical representation of the instrument. The standard method will open a new tab in a browser with a 3D view of the instrument.

input	type	explanation
parameters	dict	Dictionary with input parameters and their values
format	str	"web-gl" provides 3D tab in browser, "window" opens window with 2D views

The default value for the `format` is "web-gl"

```
1 data = detector.show_instrument(format="window", parameters={"height": 0.8})
```

## class McStasData

*Holds a single McStas data set in either 1D or 2D*

A class to handle data from McStas simulations in a transparent way which provides easy access to manipulation of the data. The included data is located in the following variables

variable	type	explanation
Intensity	float array	Numpy array containing intensity
Error	float array	Numpy array containing error on intensity
Ncount	int array	Numpy array containing number of rays in each pixel
xaxis	float array	Numpy array of xaxis if data is one dimensional
metadata	metadata class	Contains necessary meta data
plot_options	plot_options class	Preferences for plotting the data

## McStasData method set\_xlabel

In [22]: Instr.interface()

energy	<input type="text" value="10.0"/>	// [meV] Energy of source
delta_energy	<input type="text" value="8.0"/>	// [meV] Energy spread of source
rotation_y	<input type="text" value="180.0"/>	// [deg] Rotation around vertical
rotation_x	<input type="text" value="0.0"/>	// [deg] Rotation around horizontal
material	<input type="text" value="Pb"/>	// Material choice for extra material sample

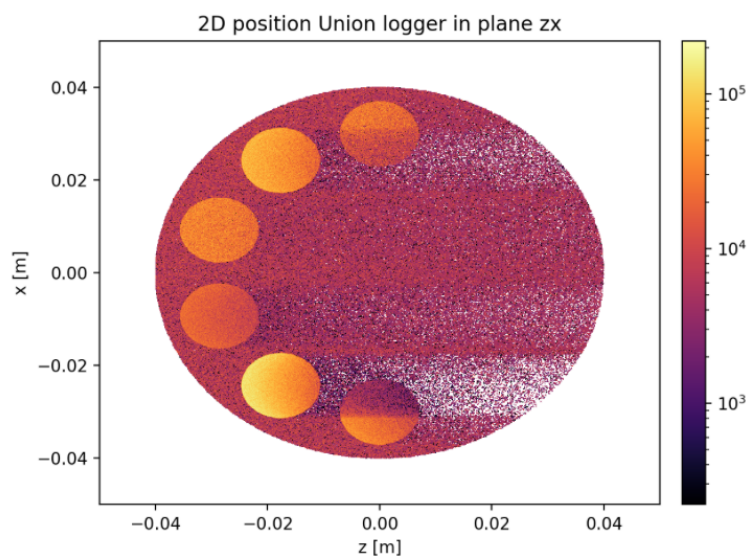
Run

ncount

mpi

☐ Live results

Figure 3



Choose monitor

Plot options

Log plot ☒

Orders of magnitude

Colormap category

Colormap

Figure 1: McStasScript widget interface for performing simulations.

*Sets the xlabel of a data set*

Method for setting xlabel on a data set, similar methods exists for ylabel and title with same syntax.

```
1 data[0].set_xlabel("custom xlabel [m]")
```

## McStasData method set\_plot\_options

*Sets plotting preferences for data set*

Plotting options are associated with the data set instead of being given during the plotting. All plot options are given as a dictionary input. Currently the following are available.

name	type	explanation
log	bool or int	plot on logarithmic scale
orders_of_mag	float	maximum orders of magnitude for colorscale
colormap	string	name of colormap to be used (matplotlib library)
show_colorbar	bool	default True, which shows colorbar on 2D plots
cut_max	float	cut top of data, 1 is all data
cut_min	float	cut bottom of data, 0 includes is all data
left_lim	float	lower limit of plot
right_lim	float	higher limit of plot
top_lim	float	top limit (Only 2D)
bottom_lim	float	bottom limit (Only 2D)
x_axis_multiplier	float	Multiplier for xaxis, for example change unit
y_axis_multiplier	float	Multiplier for yaxis, for example change unit

```
1 data[0].set_plot_options(log=True, colormap="hot")
```

It is often simpler to access the data using the name of the monitor rather than the index, which can be done using the function *name\_search*. The function will also find the data if the filename is given instead of the component name. In this example data is a list of McStasData objects.

```
1 PSD_sample = functions.name_search("PSD_sample", data)
2 PSD_sample.set_plot_options(log=True, colormap="hot")
```

Since setting the plot options will be a very frequent operation, a function is provided for this particular operation.

```
1 functions.name_plot_options("PSD_sample", data, log=True, colormap="hot")
```

In most circumstances McStasData objects will be returned from simulations performed with McStasScript. It is however possible to load a data folder that contains a mccode.sim file and the associated data. The returned data is a list of McStasData objects.

```
1 data = functions.load_data("data_folder_name")
```

## function make\_plot

*plots single McStasData object or a list of these objects*

Function for simple plotting of McStasData objects.

input	type	explanation
first argument	McStasData array	data to be plotted
figsize	tuple of two floats	size of figure in inches, (width, height)
no_data_to_black	bool	Default False where missing data is white (0 on log plots)
filename	string	If provided the plot will be saved to disk with given filename

Here all data in the list of data is plotted according to the preferences stored in the *plot\_options* class of each data set. Here the figure size is set to 10 by 8 inches.

```
1 plotter.make_plot(data, figsize=(10, 8))
```

It is also possible to use any keyword arguments from the *set\_plot\_options* method to modify the plotting.

```
1 plotter.make_plot(data, log=True, show_colorbar=False)
```

When a list of data objects is plotted, it is possible to specify the options for the individual plots as a list for the keyword argument. Here the first dataset would be plotted with the *log* option, while the second would be plotted normally.

```
1 plotter.make_plot(data, log=[True, False], orders_of_mag=5)
```

When using the keyword arguments from *set\_plot\_options*, the *plot\_options* of each dataset are applied, so if the data is plotted again these options are remembered.

## function make\_sub\_plot

*plots single McStasData object or a list of these as subplots*

Class for simple plotting of McStasData objects in one window. Will be expanded over time to contain more control over the resulting plots. Currently only the initialization is done so the returned object has no useful methods.

input	type	explanation
first argument	McStasData array	data to be plotted
figsize	tuple of two floats	size of figure in inches, (width, height)
no_data_to_black	bool	Default False where missing data is white (0 on log plots)
filename	string	If provided the plot will be saved to disk with given filename

Here all data in the list *data* is plotted according to the preferences stored in the *plot\_options* class of each data set. As with *make\_plot*, keyword arguments from *set\_plot\_options* can be used to affect all or some plots.

```
1 plotter.make_sub_plot(data, log=[True, True, False])
```

## function interface

*Plots a list of McStasData as a widget*

The plotter module has a interface function that will provide a widget interface when used in a jupyter notebook. It is possible to customize the plot and choose which monitor to view. Requires use of matplotlib widget mode.

```
1 %matplotlib widget
2 plotter.interface(data)
```

## class McStas\_file

*Class for working with existing McStas files*

This class can read a McStas instrument file and produce either a McStasScript *McStas\_instr* object or McStasScript python file that includes the information contained within the instrument file. The class is initialized with the name of the McStas file to be read. When using this system it is highly recommended to check that two simulations produce the same output when given identical parameters.

input	type	explanation
filename	str	McStas instrument filename

```
1 Reader = reader.McStas_file("ILL-IN5.instr")
```

### McStas\_file method add\_to\_instr

*Adds the content of the McStas file to the provided McStas\_instr instance*

This method is used when a McStas\_instr instance of a McStas file is required.

input	type	explanation
Instr	McStas_instr	McStas_instr instance

Here the defined Reader is used to produce a McStas\_instr instance called IN5.

```
1 IN5 = instr.McStas_instr("IN5")
2 Reader.add_to_instr(IN5)
```

### McStas\_file method write\_python\_file

*Writes python file that describes a given instrument using McStasScript*

This method will write a python file that describes the original instrument using McStasScript. This is useful if a project was started in a instrument file, but should be converted to McStasScript. It is always simple to convert back to a instrument file using the *write\_full\_insttument* method.

input	type	explanation
filename	str	filename of produced python file
force	boolean	True if the python file should be overwritten

Here the defined *Reader* is used to write a python file capable of reproducing the instrument. It is selected not to overwrite any existing file for safety (which is the default).

```
1 Reader.write_python_file("IN5.py", force=False)
```

## 5.1 Advanced use

The parts of the api covered by the documentation so far is the simplest way of using the API, but some additional methods in the McStas\_instr are useful for experienced python users that want more direct access to the underlying classes.

### McStas\_instr method get\_component

*Returns the component class instance of a selected component*

It is possible to get direct access to the component instances inside the McStas\_instr instance for direct manipulation. This can make the syntax a bit shorter in some cases.

```
1 guide_piece = detector.get_component("beam_extraction")
```

### McStas\_instr method get\_last\_component

*Returns the component class instance of the last component in the component sequence*

Same as *get\_component* but no argument is needed when returning the last component of the sequence.

```
1 guide_piece = detector.get_last_component()
```

## class component

*Holds information on a component and methods for updates and writing to file*

The *component* class is used as a superclass for each component type added to the instrument. The subclass for a specific component type also includes attributes for each parameter of the component, and these can be changed directly. The class is frozen after initialize so no new attributes can be created, and in this way misspelled parameter names are caught on user input. Most of the methods contain in the component class are just passed directly to the *McStas\_instr* and thus does not require further explanation, they are however listed here for completeness.

### component method show\_parameters

*Equivalent to component\_help in McStas\_instr, also shows changed parameters*

### component method show\_parameters\_simple

*Same information as show\_parameters, but without use of ANSI colors*

### component method set\_AT

*Equivalent to set\_component\_AT in McStas\_instr*

### component method set\_ROTATED

*Equivalent to set\_component\_ROTATED in McStas\_instr*

### component method set\_RELATIVE

*Equivalent to set\_component\_RELATIVE in McStas\_instr*

### component method set\_parameters

*Equivalent to set\_component\_parameter in McStas\_instr*

### component method set\_comment

*Equivalent to set\_component\_comment in McStas\_instr*

### component method \_freeze

*Freezes the object, an error will occur if new attributes are added*

### component method \_unfreeze

*Unfreezes the object, new attributes can be added*

### component method write\_component

*Writes the component to file*

input	type	explanation
first argument	file identifier	file identifier ready for writing

### component method print\_long

*Prints information on the component to the terminal*



## 6 Discussion

This section contains discussion on the python module.

### 6.1 Possible improvements / requests

Features that are still missing and should be added. Also keeps track of user requests.

#### 6.1.1 Limits on parameters

Allow user to easily set limits on parameters and generate appropriate input sanitation for instrument file with error message.

#### 6.1.2 Methods for removing parameters / variables / components

When using the software from a terminal it could be useful to remove components. Might also be useful to be able to move a component to another location in the sequence.

### 6.2 Jupyter notebook experience

It is entirely possible to write an instrument file from a jupyter notebook using this tool, but at this point it behaves more like a script, and thus there is no inherent benefit. The main issue is that rerunning a cell will cause errors because the same components are added again, and they recognize the names are not unique. Should instead allow to update components when the same name is used, but this adds a severe risk of users replacing an earlier component instead of creating a new.

Another issue is the lack of feedback beyond printing all added components. A simple improvement would be to have a method that prints all changes since last print was executed, which would be a natural end of each cell.

If a component instance is already defined, syntax completion does work with all parameters which is beneficial. It does however cause users to create two cells for each component to be added, as the component need to be initialized before syntax completion works.