

# McStasScript

Mads Bertelsen

May 10, 2019

## 1 Introduction

This document serves as the documentation for the McStasScript scripting language for python. Its purpose is to generate McStas instrument files from python which is simply another way of writing an instrument file. The main advantages is 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.

## 2 Documentation

This section describes the classes and their methods.

**class McStas\_instr**

*Holds methods for creating a McStas instrument file*

Initiating an instance of the class requires a name to be given as the first argument and two optional keyword arguments currently supported, allowing information on the author an origin of the code. In the table below the positional arguments are above the dotted 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
mcrun_path	string	path to local mcrun
mcstas_path	string	path to mcstas directory

Below an instance called detector will contain an instrument called "LOKI\_detector" while the instance named example has a instrument named test with a specified author. Both need information on the local McStas installation in order to load components and run the final instrument.

```
1 detector = McStasScript.McStas_instr("LOKI_detector",
2                                     mcrun_path = "/usr/local/mcstas/2.5/bin",
3                                     mcstas_path = "/usr/local/mcstas/2.5")
4 example = McStasScript.McStas_instr("test", author="Mads Bertelsen",
5                                     mcrun_path = "/usr/local/mcstas/2.5/bin",
6                                     mcstas_path = "/usr/local/mcstas/2.5")
```

**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("height", value=1.0, comment="Height in [m]")
3 detector.add_parameter("string", "reflection_filename")
4 detector.add_parameter("string", "data_filename", value="\data.dat\")
```

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 *refelction\_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 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=4, value=[0.65E-6, 0.65E-6, 1E-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\_initialize

*Adds line of code to initialize section*

This methods adds a line of text 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	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")
```

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

## 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)
AT_RELATIVE	string	name of earlier component used as reference for position
ROTATED	float list[3]	rotation around (x,y,z)
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)
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", 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 here. This method has no arguments.

```

1 detector.print_components()
2 Origin           Arm           AT   [0, 0, 0]   ABSOLUTE
3 source           Source_simple AT   [0, 0, 0]   RELATIVE Origin
4 pre_guide_slit   Slit          AT   [0, 0, 1]   RELATIVE Origin
5 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 methods 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 has been registered correctly.

```
1 detector.print_component("source")
2
3 COMPONENT source = Source_simple
4   yheight = 0.12 [m]
5   xwidth = 0.12 [m]
6   E0 = energy [meV]
7 AT [0, 0, 0] RELATIVE Origin
8 ROTATED [0, 0, 0] RELATIVE Origin
```

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 are 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
first argument	float list[3]	rotation around (x,y,z)
RELATIVE	string	name of earlier component used as reference for rotation

```
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.

```
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 additional 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. Could for example be used if several guides were simulated after the source, and each of these 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 everything after JUMP in the McStas keyword line, so for example with the syntax below. Here there is no point in iterating over a guide, and merely shows the syntax.

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

### McStas\_instr method set\_component\_comment

*Method for updating the comment on a component*

It is also possible to add a comment to a component after it was defined.

### McStas\_instr method set\_component\_comment

*Method for updating the comment on a component*

It is also possible to add a comment to a component after it was defined.

```
1 detector.set_component_comment("beam_extraction", "Simulating severe misalignment")
2 detector.print_component("beam_extraction")
3 // Simulating severe misalignment
4 COMPONENT beam_extraction = Guide_gravity
5 AT [0, 0, 2] RELATIVE pre_guide_slit
6 ROTATED [0, 2.0, 0] RELATIVE pre_guide_slit
```

### 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 methods runs the simulation using the mcrun commands of the system and returns the resulting data as a array of McStasData objects. An error will occur if the fodldername already exists.

input	type	explanation
foldername	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 mcstas flags added to mcrun launch command
mcrun_path	string	absolute path to mcrun (if not already in path)

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

## 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

*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



```
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`.

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

## class `make_plot`

*plots single McStasData object or an array of these*

Class for simple plotting of `McStasData` objects. 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	<code>McStasData</code> array	data to be plotted

Here all data in the array data is plotted according to the preferences stored in the `plot_options` class of each data set.

```
1 plot = McStasScript.make_plot(data)
```

## class `make_sub_plot`

*plots single McStasData object or an array 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	<code>McStasData</code> array	data to be plotted

Here all data in the array data is plotted according to the preferences stored in the `plot_options` class of each data set.

```
1 plot = McStasScript.make_sub_plot(data, log=[1,0,1], max_orders_of_mag=[10,2,4])
```

## 2.1 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*

## 3 Discussion

This section contains discussion on the python module.

### 3.1 Possible improvements / requests

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

#### 3.1.1 Add code to trace

It should be possible to add code directly to trace, for example include statements. The position of this code should be relative to components.

#### 3.1.2 FINALLY

Should be possible to add code to the FINALLY section which is ignored so far.

#### 3.1.3 Limits on parameters

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

#### 3.1.4 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.

### 3.2 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.