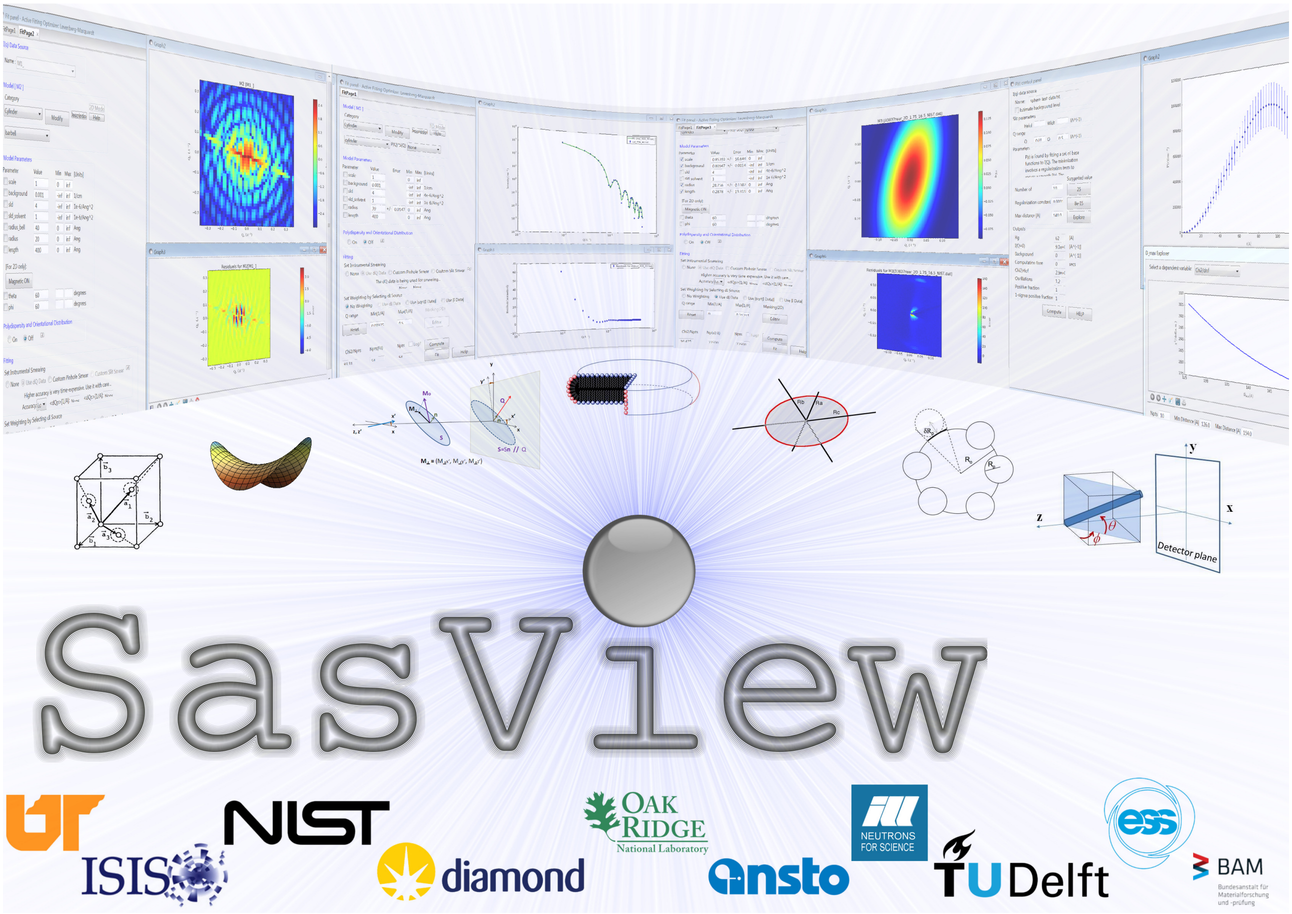
q
3


SasView
Detector plane
NIST
ISIS
diamond
Oak Ridge National Laboratory
ansto
ILL Neutrons for Science
TU Delft
ess
BAM Bundesanstalt für Materialforschung und -prüfung

SasView Tutorials

Creating Custom Fitting Models
in SasView Version 5.x

www.sasview.org

Preamble

SasView was originally developed by the University of Tennessee as part of the Distributed Data Analysis of Neutron Scattering Experiments (DANSE) project funded by the US National Science Foundation (NSF), but is currently being developed as an Open Source project hosted on GitHub and managed by a consortium of scattering facilities. Participating facilities include (in alphabetical order): the Australian National Science & Technology Centre for Neutron Scattering, the Diamond Light Source, the European Spallation Source, the Federal Institute for Materials Research and Testing, the Institut Laue Langevin, the ISIS Pulsed Neutron & Muon Source, the National Institute of Standards & Technology Center for Neutron Research, the Oak Ridge National Laboratory Neutron Sciences Directorate, and the Technical University Delft Reactor Institute.

SasView is free to download and use, including for commercial purposes.

If you make use of SasView

If you use SasView to do productive scientific research that leads to a publication, we ask that you acknowledge use of the program with the following text:

> *This work benefited from the use of the SasView application, originally developed under NSF Award DMR-0520547. SasView also contains code developed with funding from the EU Horizon 2020 programme under the SINE2020 project Grant No 654000.*

Contributors to this Tutorial

Steve King (stephen.king@stfc.ac.uk)

Paul Butler (butlerpd@udel.edu)
Wojciech Potrzebowski (Wojciech.Potrzebowski@ess.eu)
Annika Stellhorn (annika.stellhorn@ess.eu)

Revisions

Last revised: 22 September 2023

Learning Objective

This tutorial will demonstrate the various ways it is possible to extend the base library of Fitting Models (ie, Form Factors) shipped with SasView or those available for download as Plugin Models from the *Model Marketplace* (http://marketplace.sasview.org/).

The examples in this tutorial are presented in order of increasing sophistication and the degree of programming competency (mainly in the Python language) that is required to implement them. Indeed, Example 1 does not require any programming knowledge at all!

Whilst each of the examples have been written to stand alone, readers are strongly encouraged to work their way through the complete set.

The program interface shown in this tutorial is SasView Version 5.0.5 running on a Windows platform but, apart from a few small differences in look and functionality, this tutorial is generally applicable to SasView 5.x running on any platform.

Contents

Running SasView

Windows

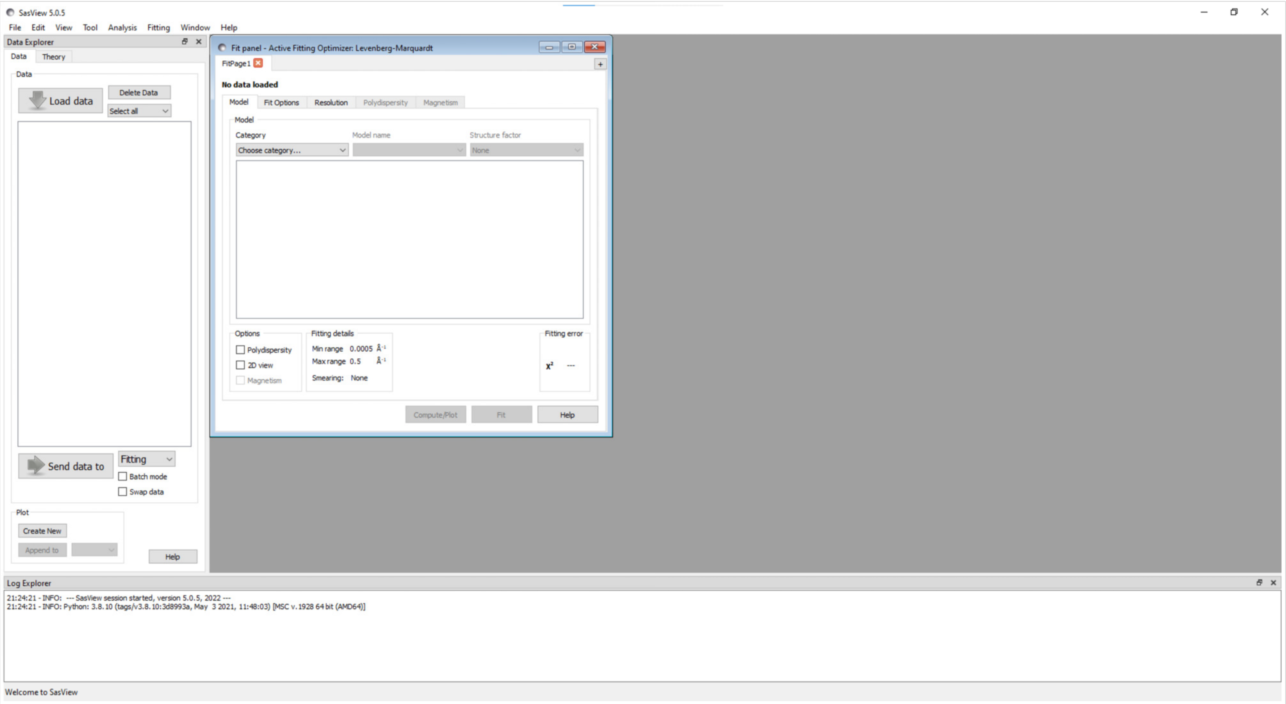
Either select SasView from the ‘**Start Menu**’ icon (or ‘**Start**’ > ‘**All Programs**’ if using Windows 7) or, if you asked the installer to create one, double-click on the SasView desktop icon.

SasView

Mac OS

Go in to your ‘**Applications**’ folder and select SasView.

The SasView main window will open.



No data loaded

Min range 0.0005 Å⁻¹
Max range 0.5 Å⁻¹
Smearing: None

χ^2 ---

```
21:24:21 - INFO:  --- SasView session started, version 5.0.5, 2022 ---
21:24:21 - INFO: Python: 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)]
```

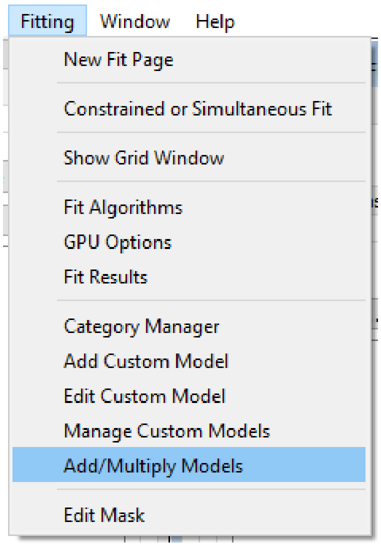
Welcome to SasView

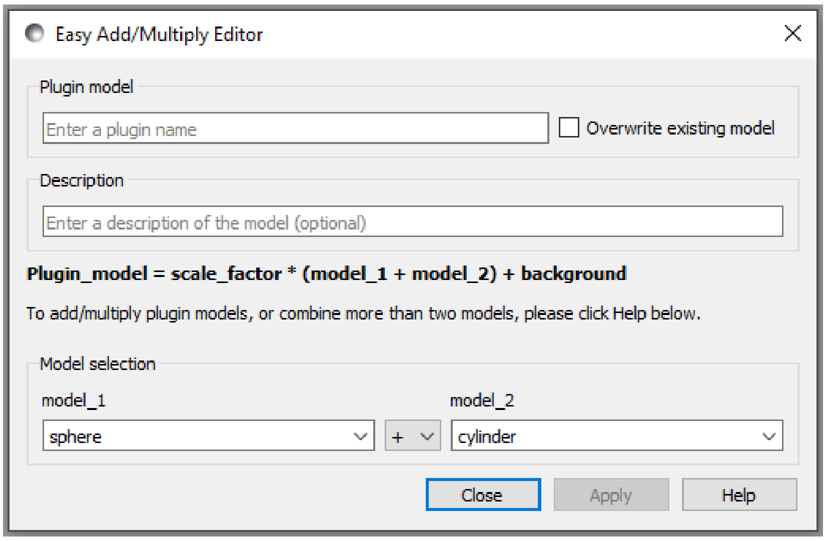
Tip: If you are not yet familiar with SasView, now would be a good time to read the tutorial ‘*Getting Started with SasView*’!

Example 1 – Combining Two Library Models

This is a common use case. Whilst the SasView FitPage allows you to combine a Model (ie, Form Factor) with a Structure Factor, it does not allow you to combine two Models. Instances where this might be necessary include where the scattering of interest is superimposed on a power law background (as often happens when studying porous systems) or where the scattering of interest arises from objects of different shape (eg, spherical and ellipsoidal micelles) or, perhaps, where the system is bimodal. In all of these examples, fitting a single Model is unlikely to suffice. This is where the ***Add/Multiply Editor*** *may be of use.*

Go to the Menu Bar and select **Fitting** > **Add/Multiply Models**. This brings up the *Easy Add/Multiply Editor*. This editor will create a custom Plugin Model for SasView to use.




Easy Add/Multiply Editor
Plugin model
Enter a plugin name
☐ Overwrite existing model
Description
Enter a description of the model (optional)

Plugin_model = scale_factor * (model_1 + model_2) + background

To add/multiply plugin models, or combine more than two models, please click Help below.

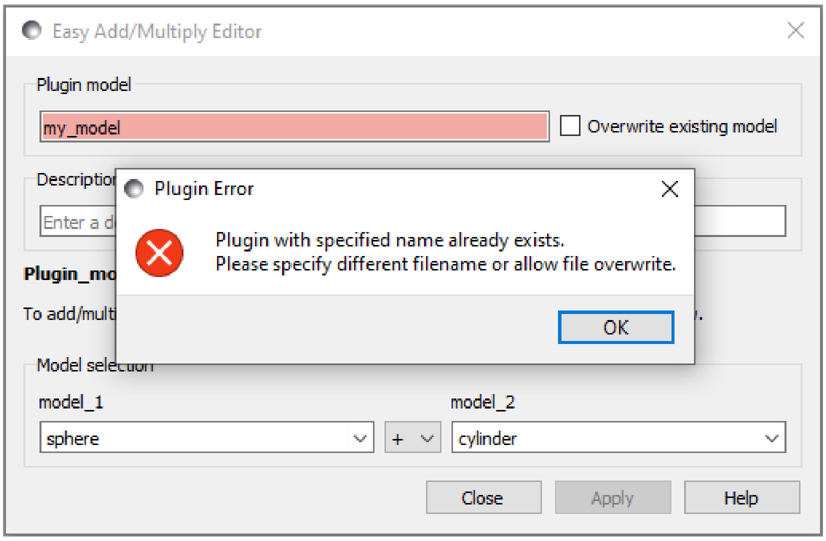
Model selection

model_1: sphere

+

model_2: cylinder

Provide a <u>unique</u> name for the Plugin Model about to be created. This name must not contain any spaces (use underscores to separate words if necessary).



my_model

Plugin with specified name already exists.
Please specify different filename or allow file overwrite.

If the name of the Plugin Model is not unique, for example, because it is the name of a library Model shipped with SasView, or that of an existing Plugin Model, you will be warned as shown above. In this situation you either need to provide a different name or check the box to allow the existing Model to be overwritten. It is unwise to overwrite library models or create models with the same names as library models!

Now add a short description of the Plugin Model.

Then select the two Models to be combined and, in between those drop-down boxes, also select the manner in which the Models are to be combined. There are two options:

+ **used to linearly combine the two models;**
plugin_model = scale_factor * (P(Q)_1 + P(Q)_2) + background

@ **used to multiply a form factor by a structure factor;**
plugin_model = scale_factor * (P(Q) @ S(Q)) + background

Tip: There is also the option to use the normal multiply operator *, to generate models of the form:

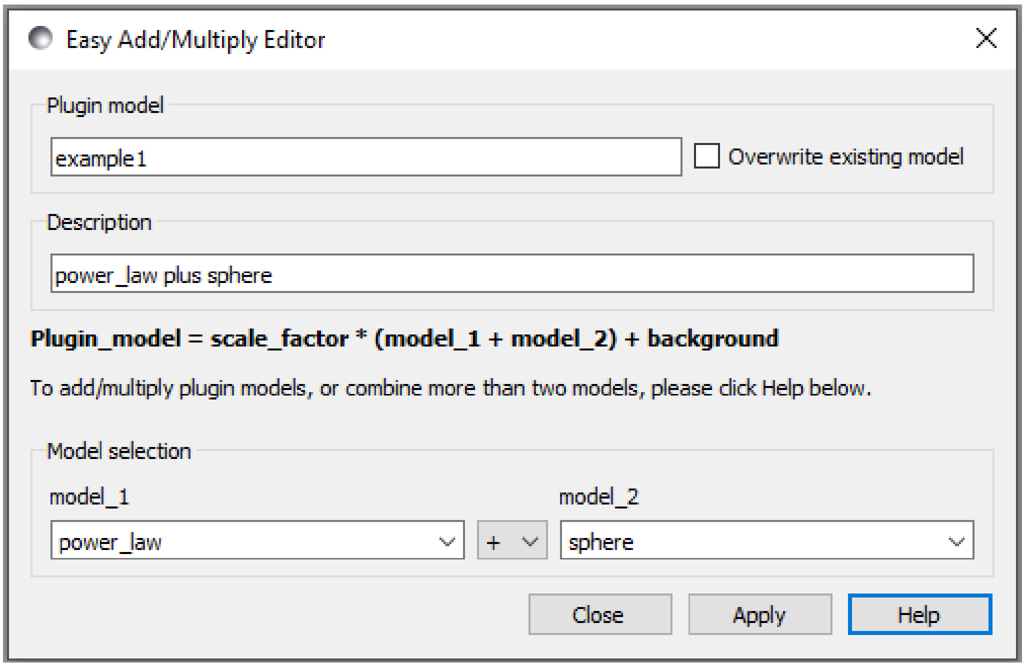
plugin_model = scale_factor * (P(Q)_1 * P(Q)_2) + background

or

plugin_model = scale_factor * (P(Q) * S(Q)) + background

however, these should never be necessary. The first, multiplying two Form Factors together, would be a highly unusual requirement, whilst the second duplicates functionality available through the @ operator anyway. Use of the @ operator is discussed further in **Example 2**.

Finally, click **Apply.** Here is an example combining the *power_law* and *sphere* Models.



example1

☐ Overwrite existing model

power_law plus sphere

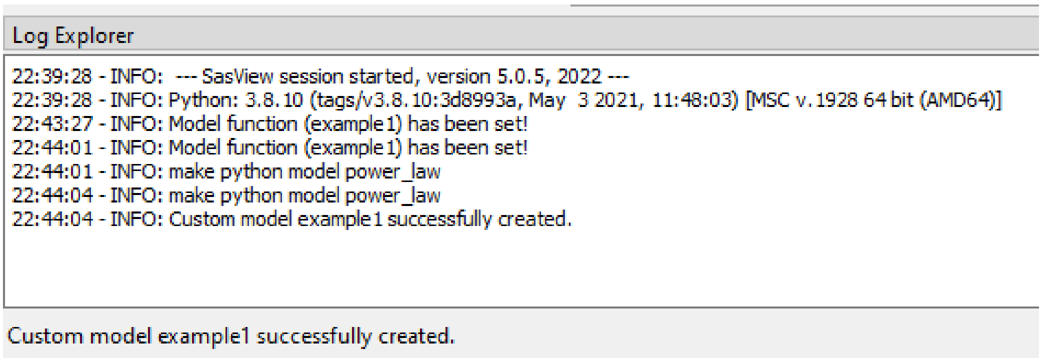
Plugin_model = scale_factor * (model_1 + model_2) + background

To add/multiply plugin models, or combine more than two models, please click Help below.

model_1: power_law

+

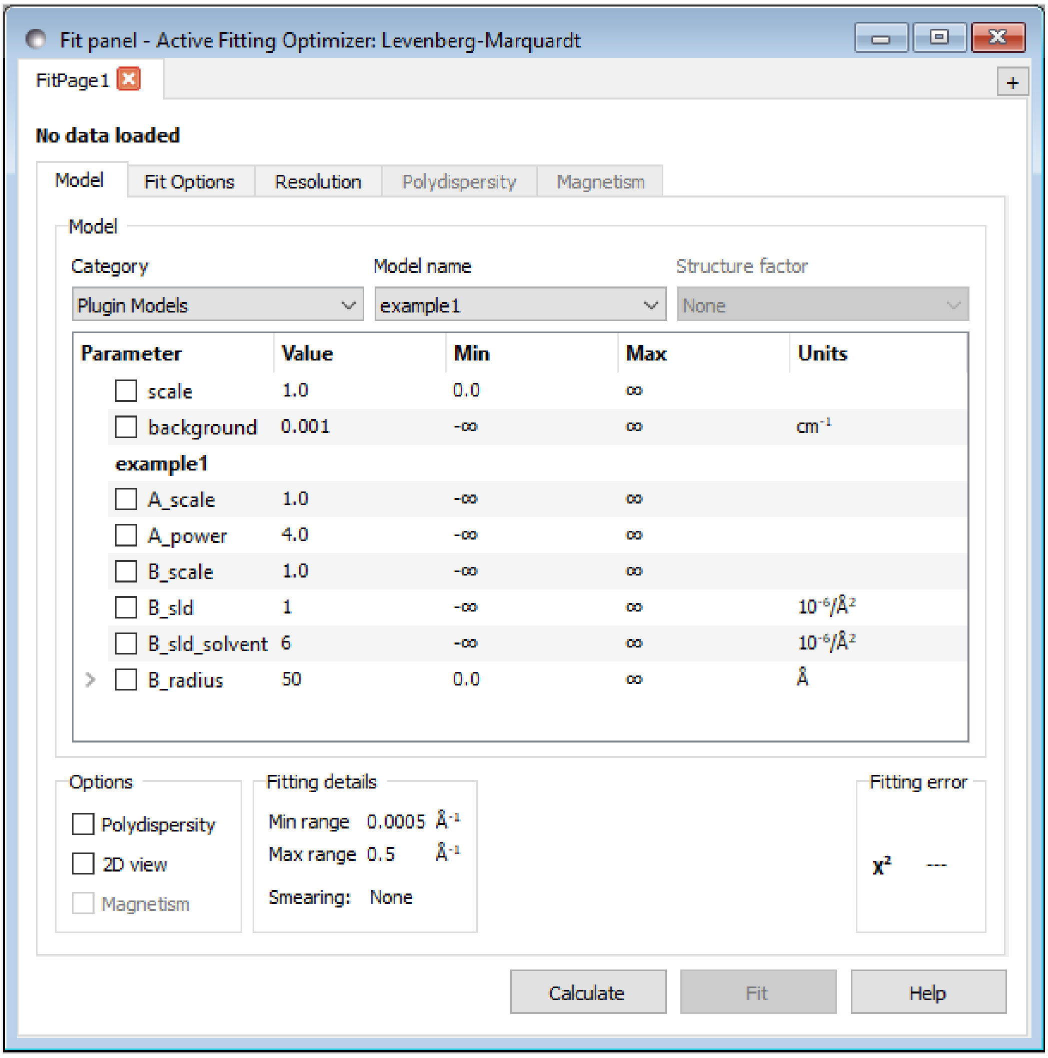
model_2: sphere



```
22:39:28 - INFO:  --- SasView session started, version 5.0.5, 2022 ---
22:39:28 - INFO: Python: 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)]
22:43:27 - INFO: Model function (example1) has been set!
22:44:01 - INFO: Model function (example1) has been set!
22:44:01 - INFO: make python model power_law
22:44:04 - INFO: make python model power_law
22:44:04 - INFO: Custom model example1 successfully created.
```

Custom model example1 successfully created.

To load this Plugin Model, open a FitPage, select the category Plugin Models, and then select the Model *example1*.



No data loaded

Model

Category: Plugin Models

Model name: example1

Structure factor: None

Parameter	Value	Min	Max	Units
☐ scale	1.0	0.0	∞	
☐ background	0.001	-∞	∞	cm^{-1}
example1				
☐ A_scale	1.0	-∞	∞	
☐ A_power	4.0	-∞	∞	
☐ B_scale	1.0	-∞	∞	
☐ B_sld	1	-∞	∞	10^{-6}/Å^2
☐ B_sld_solvent	6	-∞	∞	10^{-6}/Å^2
> ☐ B_radius	50	0.0	∞	Å

Options

☐ Polydispersity

☐ 2D view

☐ Magnetism

Fitting details

Min range 0.0005 Å^{-1}

Max range 0.5 Å^{-1}

Smearing: None

Fitting error

χ^2 ---

Notice how the parameters for the selected model_1 are prefixed by ‘A_’ and the parameters for the selected model_2 are prefixed by ‘B_’.

Also notice that separate *scale* parameters are introduced for each of the constituent Models. Thus the function that will actually be computed is:

intensity = *scale* * [(*A_scale* * *model_1*) + (*B_scale* * *model_2*)] + *background*

Finally, also note that the option to multiply the Plugin Model by a Structure Factor in the FitPage is denied. This restriction may be lifted in a future version of SasView. In the meantime, **Example 2** below shows how to workaround this.

Example 2 – Combining More Than Two Models

A typical use case here would be for combining a Plugin Model with a Structure Factor, but it could equally involve combining multiple Models (ie, multiple Form Factors).

Create a Plugin Model with the *Easy Add/Multiply Editor* (see **Example 1** above).

Plugin Models reside in a plugins folder in your profile folder:

(Windows) `C:\Users\<username>\.sasview\plugin_models`
(Mac) `~/.sasview/plugin_models`

Navigate to this folder and open the Plugin Model in a text editor. It will have a `.py` file extension (signifying it is a Python file).

Here is how the contents of the Plugin Model *example1.py* created in **Example 1** appear:

```
from sasmodels.core import load_model_info
from sasmodels.sasview_model import make_model_from_info

model_info = load_model_info('power_law+sphere')
model_info.name = 'example1'
model_info.description = 'power_law plus sphere'
Model = make_model_from_info(model_info)
```

To repurpose this Plugin Model it is only necessary to edit the three lines highlighted above in green. For example, to add a *hardsphere* Structure Factor to this Plugin Model change it to read:

```
from sasmodels.core import load_model_info
from sasmodels.sasview_model import make_model_from_info

model_info = load_model_info('power_law+sphere@hardsphere')
model_info.name = 'example2'
model_info.description = 'power_law plus sphere x hardsphere'
Model = make_model_from_info(model_info)
```

The changes are highlighted in yellow.

Tip: To be valid, the *load_model_info* string must only contain Model names and the operators +, @ or * (see **Example 1** above). In particular, brackets are not permitted. This means that a plugin model of the form, for example:

```
'(sphere+cylinder)@hardsphere'
```

will not work, but this form does:

```
'sphere@hardsphere+cylinder@hardsphere'
```

However, in this latter case it will be necessary to use manual constraints to control the behaviour of the structure factor *radius_effective_mode* (see below and **Example 6**).

Tip: When the @ operator is used, a *'radius_effective_mode'* drop-down box is added to the model parameter table allowing the User to select how the radius (or one of a number of other size parameters depending on the shape of the object) in the Form Factor and the effective interaction radius in the Structure Factor are to be related.

Selecting *'unconstrained'* means the two parameters will be optimised independently and it is up to the User to ensure that they remain physically realistic. Selecting any of the other options constrains the two parameters to have the same value and only the Form Factor parameter will optimize.

Furthermore, for some models a *structure_factor_mode'* drop-down will also be added. The default for this is *'P*S'*; ie, the normal case of P(Q) * S(Q). But selecting *'P*(1+beta*(S-1))'* will apply the 'beta-decoupling approximation'.

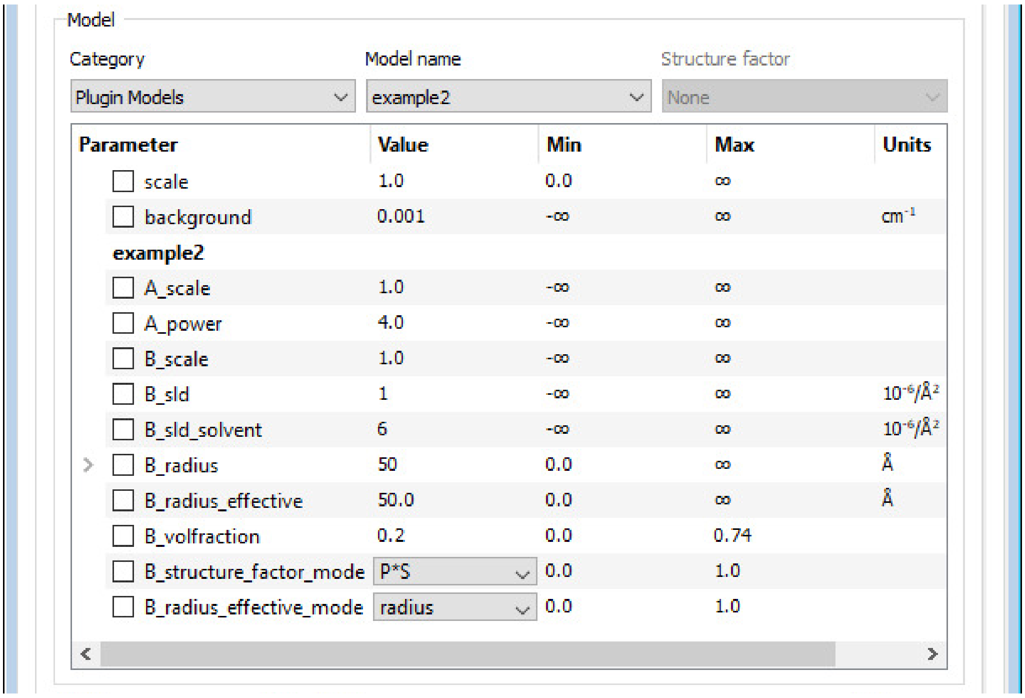
For further information, see:
https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/fitting_sq.html

Save the changed file in the plugins folder, here as *example2.py.*

The new Plugin Model will be available in SasView if:

- the program is restarted, or
- a new FitPage is opened, or
- you navigate **Fitting** > **Edit Custom Model** > Load Plugin… > highlight the model > Open > Save > Cancel.

When this Plugin Model is loaded the additional Structure Factor parameters become apparent (compare the figure below with the previous figure in **Example 1**).



Category: Plugin Models

Model name: example2

Structure factor: None

Parameter	Value	Min	Max	Units
☐ scale	1.0	0.0	∞	
☐ background	0.001	-∞	∞	cm^{-1}
example2				
☐ A_scale	1.0	-∞	∞	
☐ A_power	4.0	-∞	∞	
☐ B_scale	1.0	-∞	∞	
☐ B_sld	1	-∞	∞	$10^{-6}/Å^2$
☐ B_sld_solvent	6	-∞	∞	$10^{-6}/Å^2$
› ☐ B_radius	50	0.0	∞	Å
☐ B_radius_effective	50.0	0.0	∞	Å
☐ B_volfraction	0.2	0.0	0.74	
☐ B_structure_factor_mode	P*S	0.0	1.0	
☐ B_radius_effective_mode	radius	0.0	1.0	

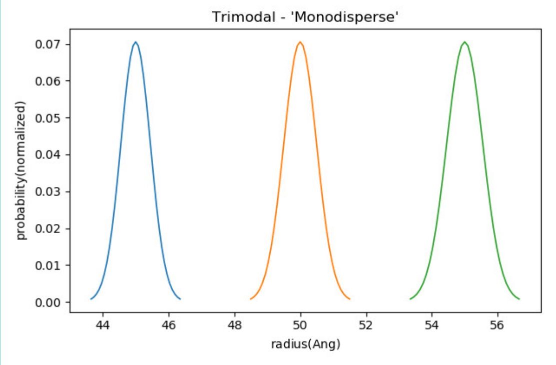
Similarly, if you knew you had, say, a trimodal system of spherical particles you might need a model like this:

```
from sasmodels.core import load_model_info
from sasmodels.sasview_model import make_model_from_info

model_info = load_model_info('sphere+sphere+sphere')
model_info.name = 'example2b'
model_info.description = 'trimodal spheres'
Model = make_model_from_info(model_info)
```

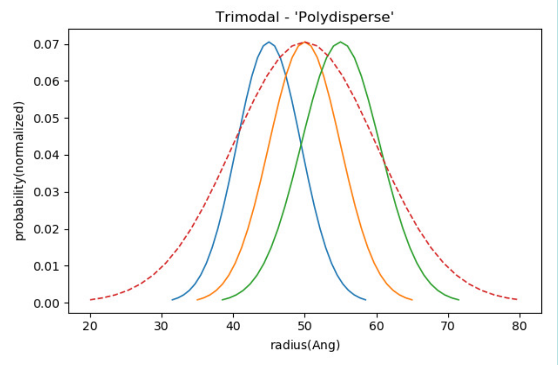
Aside: Modality and size polydispersity are not the same thing, although it may sometimes be difficult to distinguish between them!

A sample is multi-modal if it contains two or more *distinct* sub-populations of scattering objects. If you were to mix a dispersion of 50 Å particles with a dispersion of 100 Å particles that would create a bimodal dispersion.

Size polydispersity is where a population of scattering objects have a distribution of sizes about some significant value. The sub-populations in a multi-modal sample may or may not *also* exhibit size polydispersity. But, obviously, if the width of the respective size distributions are comparable to the difference in size between the sub-populations it may be very difficult to resolve those individual sub-populations as shown in the figures below.



Simulated size distributions for a trimodal system of 45, 50 & 55 Å particles each exhibiting 1% Gaussian size polydispersity.



The same trimodal system but now with 10% Gaussian size polydispersity. Also shown (dashed line) is a *unimodal* size distribution with 20% size polydispersity.

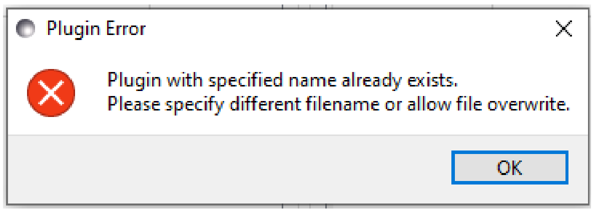
Of course, it is always possible, even likely, that SasView will not have a Model suitable for your needs, and that no combination of the existing models will suffice either. In this situation you will need to create a Model from first principles. For this there are four approaches you might take, as described in each of the next four examples.

Example 3 – Creating A Simple Model With The Model Editor

Go to the Menu Bar and select **Fitting** > **Add Custom Model**. This brings up the *Model Editor* dialog.

Provide a unique name for the Plugin Model about to be created. This name must not contain any spaces (use underscores to separate words if necessary).

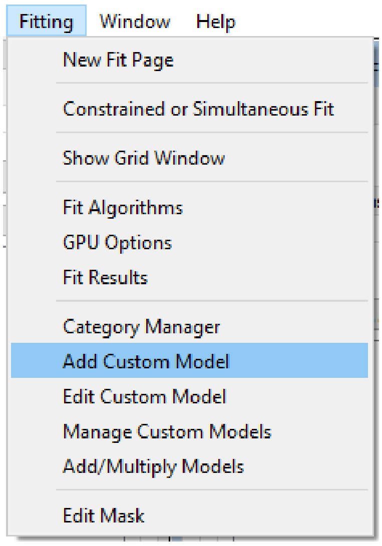
If the name of the Plugin Model is not unique, for example, because it is the name of a base Model shipped with SasView or that of an existing Plugin Model, you should be warned when you eventually click Apply as shown below.

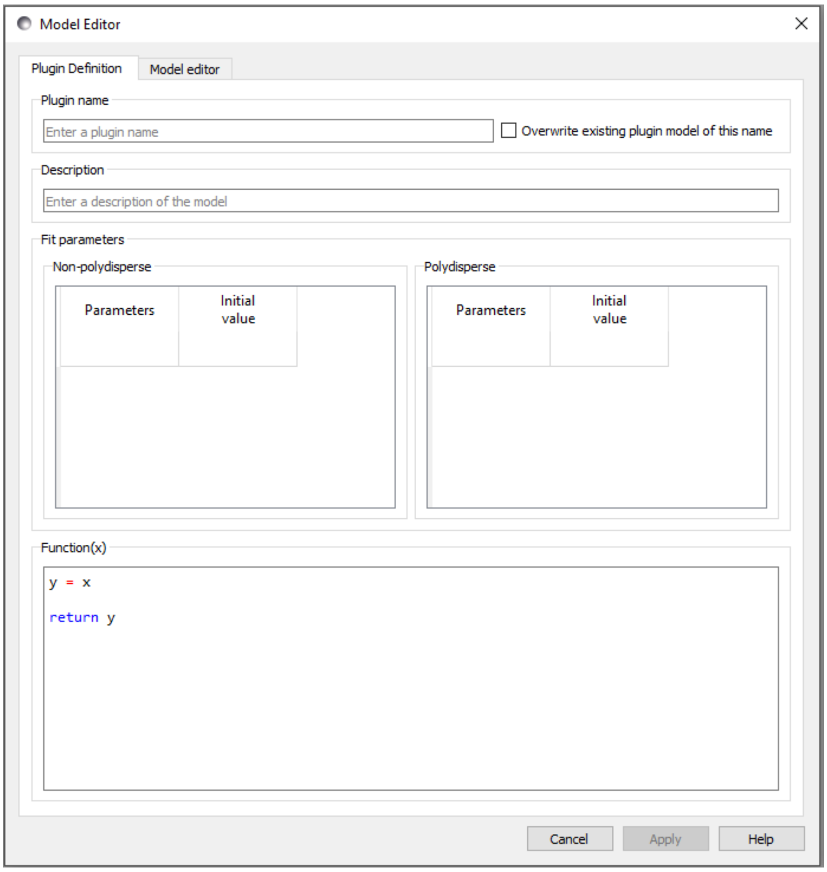


Plugin with specified name already exists.
Please specify different filename or allow file overwrite.

In this situation you either need to provide a different name or check the box to allow the existing Model to be overwritten. It is unwise to overwrite library models or create models with the same names as library models!

Now add a short description of the Plugin Model.





```
y = x

return y
```

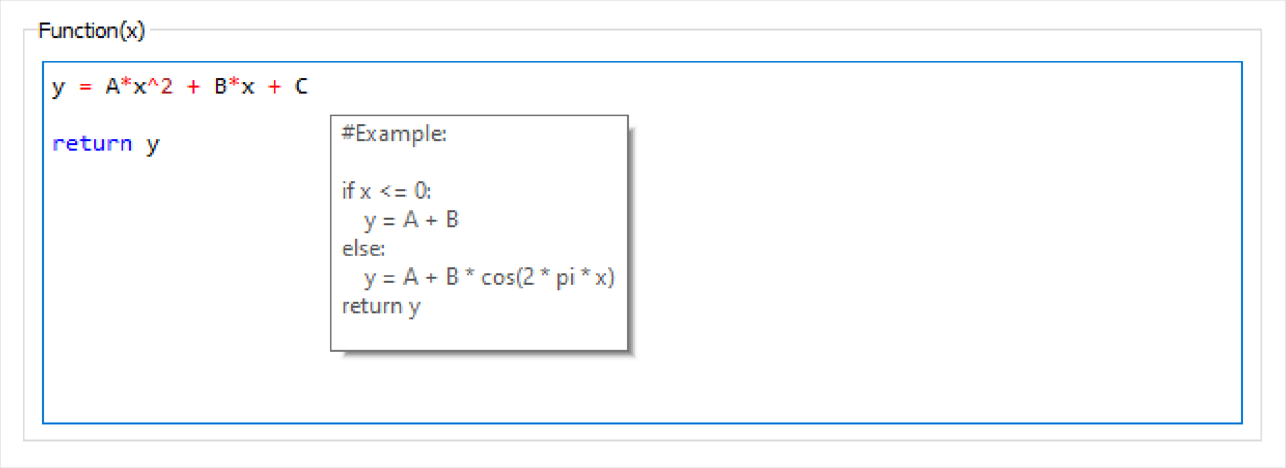
The algebraic expression for the Model to be created should be specified in the text box labelled *Function(x).* For this example, let us assume you wish to create a *parabola* Model, perhaps because you want to ascertain the contrast match point for some SANS data.

First, notice that the independent and dependent variables used by the *Model Editor* are *x* and *y* and not Q and *I*. This is deliberate, to underline that the fitting functionality in SasView is completely independent of the data it is applied to. A glance at the *Model Marketplace* (http://marketplace.sasview.org/), for example, will show that there are Models for fitting Neutron Reflectivity (NR) and Dynamic Light Scattering (DLS) data in addition to those Models intended to fit SANS/SAXS data. If you really wish to use *Q* (or *q*) or *I* (or *i*) in your function you can always assign them to *x* and *y* like this:

```
Q = x
I = your function of Q
y = I
return y
```

However, we recommend you avoid confusion and potential pitfalls by only using *x* and *y*.

Now enter the required expression. As the tooltip alludes to, this should be in valid Python. The normal mathematical hierarchy of operators also applies. So, for this example:

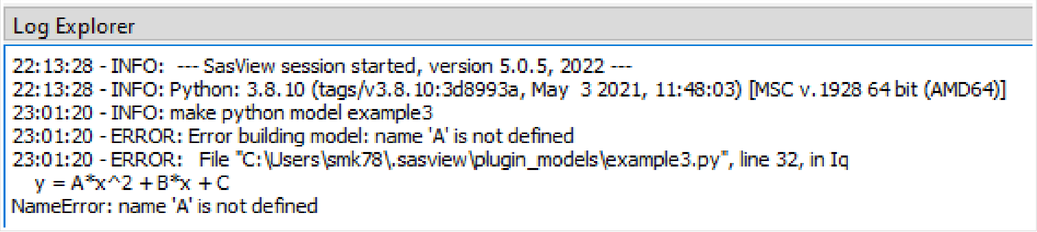


```
y = A*x^2 + B*x + C

return y
```

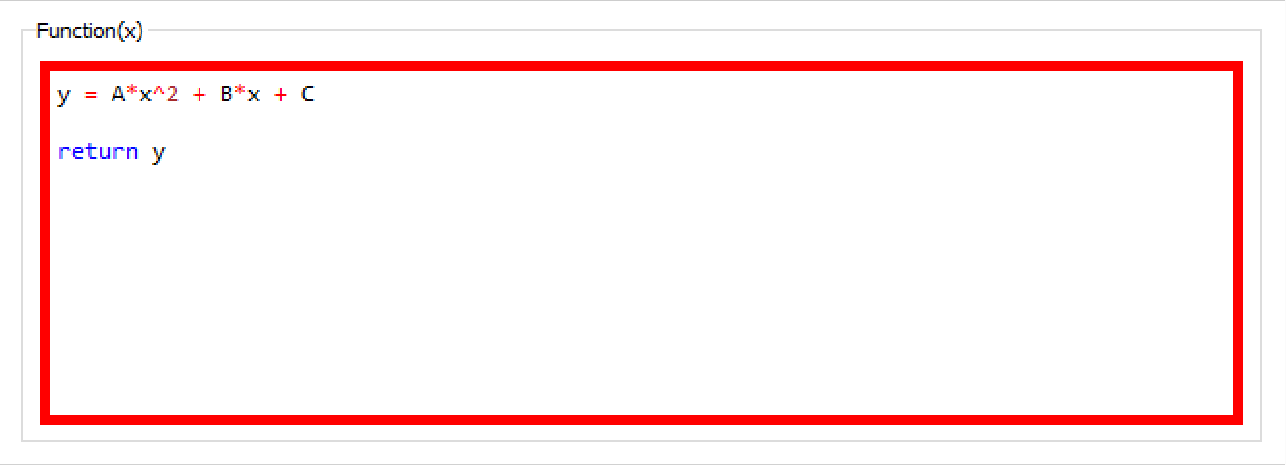
```
#Example:

if x <= 0:
   y = A + B
else:
   y = A + B * cos(2 * pi * x)
return y
```

Then click Apply.

SasView will perform some simple syntax/error checking. If it finds a problem two things happen: the border of the *Function(x)* box turns red, and an error message describing the problem is written to the *Log Explorer* at the bottom of the SasView GUI.

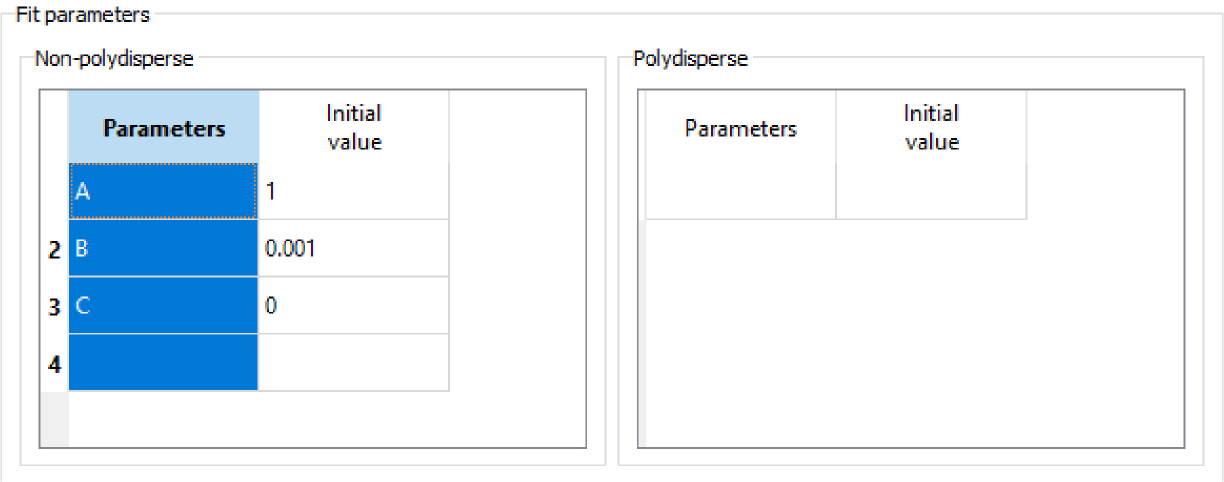


```
22:13:28 - INFO:  --- SasView session started, version 5.0.5, 2022 ---
22:13:28 - INFO: Python: 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928 64 bit (AMD64)]
23:01:20 - INFO: make python model example3
23:01:20 - ERROR: Error building model: name 'A' is not defined
23:01:20 - ERROR:   File "C:\Users\smk78\.sasview\plugin_models\example3.py", line 32, in Iq
    y = A*x^2 + B*x + C
NameError: name 'A' is not defined
```



```
y = A*x^2 + B*x + C

return y
```

In this case the check has failed because the variable *A,* and by extension the variables *B* and *C* also, have not been defined. These are the parameters that the fitting engine will optimise when the Model is used, and so need to be declared as such. To do this, click in the box labelled *Parameters*, type in the first parameter and, alongside it, enter an initial value for the parameter. Repeat the process for the remaining two parameters.



	Parameters	Initial value
	A	1
2	B	0.001
3	C	0
4		

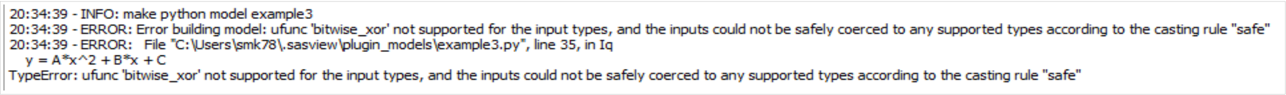
Parameters	Initial value

Notice that you have a choice of declaring parameters as ‘Non-polydisperse’ (ie, a parameter may only take one value) or ‘Polydisperse’ (ie, the parameter may take a distribution of values).

Aside: If you declare a parameter as ‘Polydisperse’ (type ‘volume’) it will also be necessary to provide a *form_volume* function (see Section 6) and to write that function **without normalizing by volume** (in order for SasView to return a number-average distribution like all other models; if you do not do this the returned distribution is a z-average distribution).

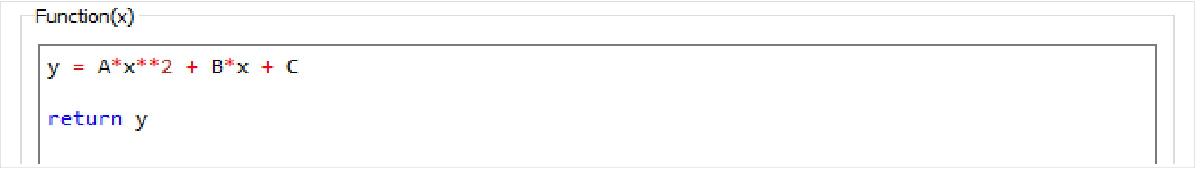
Click Apply again.

The Model check still fails, but this time with a less helpful error message:



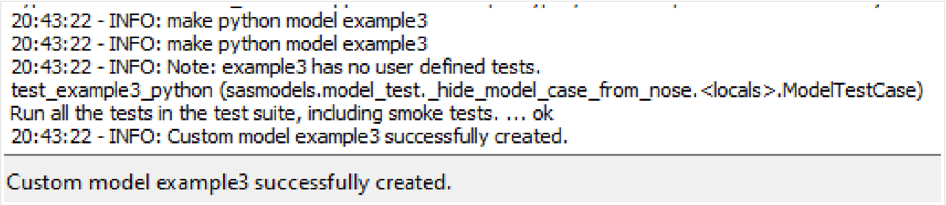
```
20:34:39 - INFO: make python model example3
20:34:39 - ERROR: Error building model: ufunc 'bitwise_xor' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule "safe"
20:34:39 - ERROR:   File "C:\Users\smk78\.sasview\plugin_models\example3.py", line 35, in Iq
    y = A*x^2 + B*x + C
TypeError: ufunc 'bitwise_xor' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule "safe"
```

The reason is that the function definition contains a non-Python operator; the ^ (caret) symbol. This operator might work in Microsoft Excel, for example, but in Python the exponentiation operator is **. So replace the incorrect operator and click Apply.



```
y = A*x**2 + B*x + C

return y
```

This time SasView confirms that the Model has been successfully created:



```
20:43:22 - INFO: make python model example3
20:43:22 - INFO: make python model example3
20:43:22 - INFO: Note: example3 has no user defined tests.
test_example3_python (sasmodels.model_test._hide_model_case_from_nose.<locals>.ModelTestCase)
Run all the tests in the test suite, including smoke tests. ... ok
20:43:22 - INFO: Custom model example3 successfully created.

Custom model example3 successfully created.
```

It does also warn that there are no unit tests for the Model, but for the purposes of this example those warnings can be ignored. However, not having unit tests is bad practice!!! See the discussion in Section 6.

Click Cancel. Open a new FitPage, load the Plugin Model *example3*, and Calculate the Model.

Notice that SasView has automatically added two additional parameters to the Model: *scale* and *background*, with the default values 1 and 0.001, respectively. SasView does this to all Models.

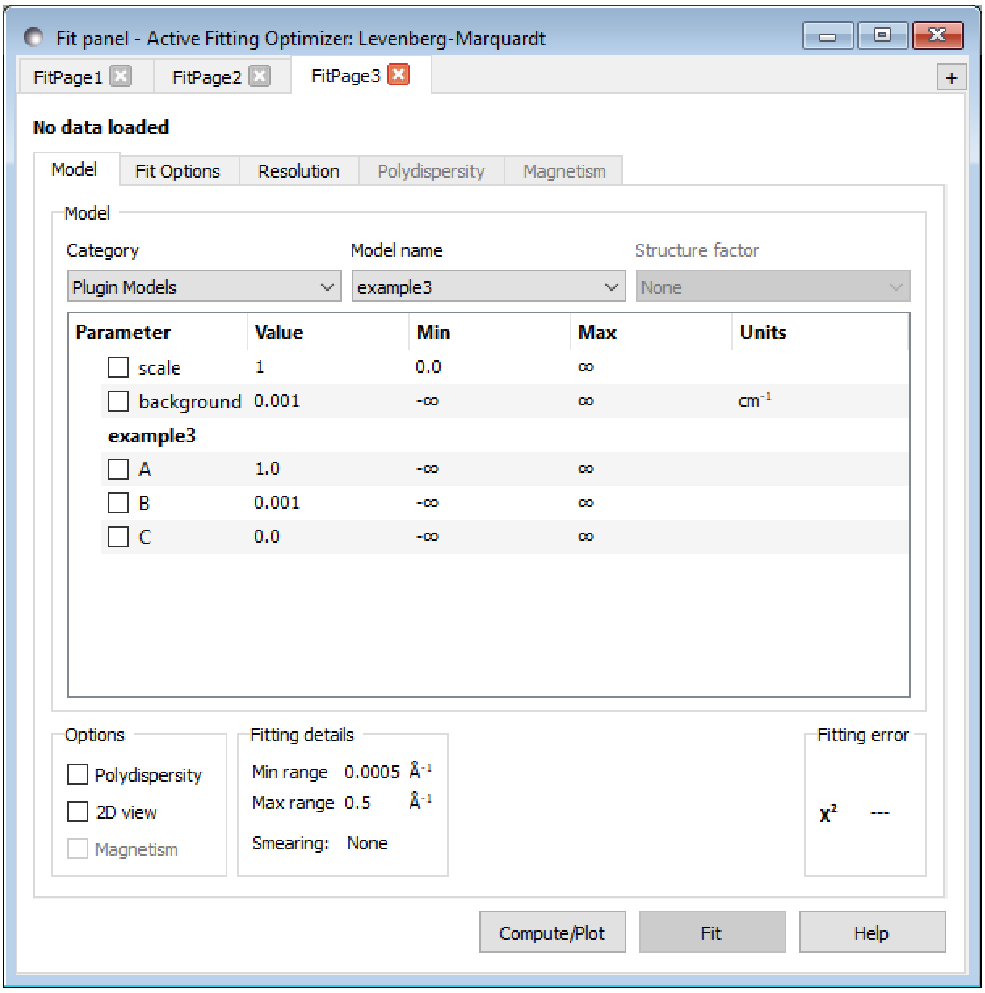
What this means in practice is that instead of computing the function you typed:

```
y = A * x**2 + B * x + C
```

SasView is actually computing the function:

```
y = scale * (A * x**2 + B * x + C) + background
```

So in order to use the Model as intended it is necessary to set *background* = 0 (and leave *scale = 1).*



No data loaded

Model

Category: Plugin Models

Model name: example3

Structure factor: None

Parameter	Value	Min	Max	Units
☐ scale	1	0.0	∞	
☐ background	0.001	-∞	∞	cm^{-1}
example3				
☐ A	1.0	-∞	∞	
☐ B	0.001	-∞	∞	
☐ C	0.0	-∞	∞	

Options

☐ Polydispersity

☐ 2D view

☐ Magnetism

Fitting details

Min range 0.0005 $Å^{-1}$

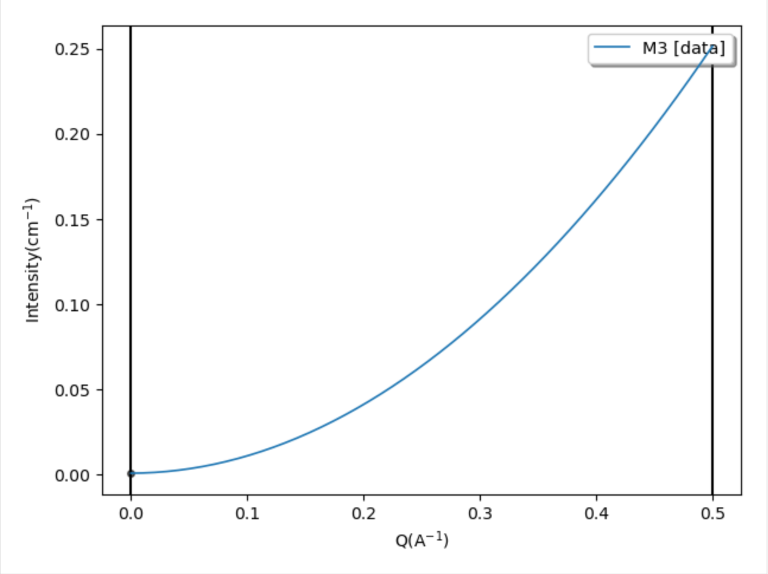
Max range 0.5 $Å^{-1}$

Smearing: None

Fitting error

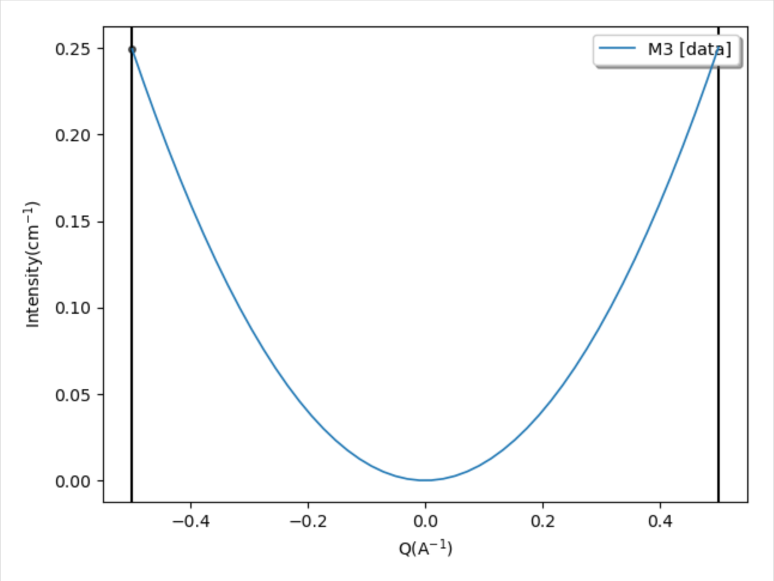
χ^2 ---

Now Compute/Plot the Model.



Notice that by default SasView labels the plot axes as *Q* and *Intensity*. These labels can be changed by right-clicking on the plot, selecting **Toggle Navigation Menu**, and then the *Edit axis…* icon 📈.

The *x*-axis range to be computed can be changed in the *Fit Options* tab on the FitPage. So, for example, by changing the minimum *Q* value the other side of the parabola can be plotted:



Aside: If required, the Python language *math* library features a comprehensive collection of mathematical functions; for further details see:
https://docs.python.org/3/library/math.html

Additional mathematical capability is available within the *numpy* (Numerical Python) and *scipy* (Scientific Python) libraries; see https://numpy.org/doc/stable/index.html and https://docs.scipy.org/doc/scipy/, respectively.

All of these libraries are included in your SasView installation and do not require separate installation. To use any functionality from them in a Model it is only necessary to ‘import’ that functionality from the relevant library. In other words:

`result = cos(0)` - to return the cosine of 0 radians

is equivalent to:

```
import math
result = math.cos(0)
```

`result = scipy.special.gamma(3)` - to return Γ(3)

is equivalent to:

```
from scipy import special
result = special.gamma(3)
```

```
if np.isfinite(result):
   print(‘result is not infinity or Nan’)
```

is equivalent to:

```
import numpy as np
if np.isfinite(result):
print(‘result is not infinity or Nan’)
```

Where possible, use *numpy* or *scipy* functions, rather than *math* functions, as the former are more likely to be GPU-compliant.

In addition, some functions commonly used by SasView have been re-written to improve their performance in the SasView environment; for further details see:
https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/plugin.html#special-functions

It should now be apparent that although this section was titled ‘Creating a Simple Model’, in actuality, the Model Editor can be used to create models of some complexity!

Example 4 – Repurposing An Existing Model

By definition, an existing Model, especially from the base library, but possibly also from the Model Marketplace, ought to contain many of the essential elements necessary in a well-written Model. These Models can therefore be used as ‘templates’ for new Models.

Locate the base library Model file *correlation_length.py* and copy it to your plugins folder. As a reminder, your plugins folder is at:

(Windows)	`C:\Users\<username>\.sasview\plugin_models`
(Mac)	`~/.sasview/plugin_models`

If you are using Windows, the base library of Fitting Models reside within the subfolder `\sasmodels\models` in the SasView installation folder (eg, `C:\SasView-5.0.5`). If you are using a Mac, it is probably easier to just search for the Model file directly.

Important! Now rename *correlation_function.py* in the plugins folder, here to *example4.py*.

Note: the only reason for choosing *correlation_function.py* here is that it is quite a clear example of the structure of a Model and also a relatively simple Model.

Open the file in a text editor of your choice. If it is a 'language-sensitive' editor that recognises Python, all the better (for example, the SasView *(Custom) Model* Editor, *Notepad*++ or *PyCharm*).

Tip: If you use the (*Custom) Model Editor* it will not be necessary to restart SasView in order to pick up the modified Model.

Below is an annotated look at the file contents as they appear in *Notepad*++:

Anything in green, preceded by a # symbol, is a comment.
Anything in orange, enclosed by """, is documentation.
Anything in grey, enclosed by " or ', is a text string.
Anything in blue is a Python statement.
Numerical values are in red.
Python function names are in pink.

The file starts with a description of the Model, written in ReSTructured text and using LaTeX to markup the equations. It is this code block that appears as the Model documentation in SasView. The Model title and parameter table are inserted automatically during the documentation build process, as is the plot of the function using the default parameters.

```
#correlation length model
# Note: model title and parameter table are inserted automatically
r"""
Definition
----------

The scattering intensity I(q) is calculated as

.. math::
    I(Q) = \frac{A}{Q^n} + \frac{C}{1 + (Q\xi)^m} + \text{background}

The first term describes Porod scattering from clusters (exponent = $n$) and
the second term is a Lorentzian function describing scattering from
polymer chains (exponent = $m$). This second term characterizes the
polymer/solvent interactions and therefore the thermodynamics. The two
multiplicative factors $A$ and $C$, and the two exponents $n$ and $m$ are
used as fitting parameters. (Respectively *porod_scale*, *lorentz_scale*,
*porod_exp* and *lorentz_exp* in the parameter list.) The remaining
parameter $\xi$ (*cor_length* in the parameter list) is a correlation
length for the polymer chains. Note that when $m=2$ this functional form
```

```
becomes the familiar Lorentzian function. Some interpretation of the
values of $A$ and $C$ may be possible depending on the values of $m$ and $n$.

For 2D data: The 2D scattering intensity is calculated in the same way as 1D,
where the q vector is defined as

.. math::  q = \sqrt{q_x^2 + q_y^2}

References
----------

#. B Hammouda, D L Ho and S R Kline, Insight into Clustering in
   Poly(ethylene oxide) Solutions, Macromolecules, 37 (2004) 6932-6937

Authorship and Verification
----------------------------

* **Author:** NIST IGOR/DANSE **Date:** pre 2010
* **Last Modified by:** Steve King **Date:** September 24, 2019
* **Last Reviewed by:**
"""
```

The next line imports two functions from the *numpy* library (see the Aside in **Example 3**).

```
from numpy import inf, errstate
```

The next four statements identify the Model and the default category it should belong to. `title` only appears in the help documentation, but `description` may appear as a tooltip within the program.

```
name = "correlation_length"
title = """Calculates an empirical functional form for SAS data characterized
by a low-Q signal and a high-Q signal."""
description = """
"""
category = "shape-independent"
```

There then follows a declaration of all the Model parameters (ie, those variables that might optimise during fitting) in a specific sequence: the parameter name, any units for the value of that parameter, a default value for that parameter, any lower and upper limits on that parameters value, the type of that parameter (see below), and a short description of that parameter.

The parameter type can be one of: “” (undeclared), “volume”, “sld” or “orientation”.

- “volume” parameters can be polydisperse;
- “sld” parameters can have magnetic moments, and;
- “orientation” parameters are used to translate orientations.

But remember, the `scale` and `background` parameters are added automatically.

```
# pylint: disable=bad-continuation, line-too-long
#             ["name", "units", default, [lower, upper], "type","description"],
parameters = [
              ["lorentz_scale", "", 10.0, [0, inf], "", "Lorentzian Scaling Factor"],
              ["porod_scale", "", 1e-06, [0, inf], "", "Porod Scaling Factor"],
              ["cor_length", "Ang", 50.0, [0, inf], "", "Correlation length, xi"],
              ["porod_exp", "", 3.0, [0, inf], "", "Porod Exponent, n"],
              ["lorentz_exp", "", 2.0, [0, inf], "", "Lorentzian Exponent, m"],
             ]
# pylint: enable=bad-continuation, line-too-long
```

The next code block defines the calculation of I(q), including any error-trapping (in this case allowing execution of the Model even if a divide-by-zero error is encountered). The function called to compute I(q) is named `Iq`, but the computed intensities are returned in the variable `inten`.
The parameters to perform the calculation must be passed to the function in the order they were declared above (which is also the order in which they will appear in the FitPage). Note that the first parameter passed must be `q`.

```
def Iq(q, lorentz_scale, porod_scale, cor_length, porod_exp, lorentz_exp):
    """
    1D calculation of the Correlation length model
    """
    with errstate(divide='ignore'):
        porod = porod_scale / q**porod_exp
        lorentz = lorentz_scale / (1.0 + (q * cor_length)**lorentz_exp)
    inten = porod + lorentz
    return inten
```

The following line:

```
Iq.vectorized = True
```

determines if the `q` values are passed to the computation kernel all in one go (True) or one at a time (False). Obviously the former is to be preferred but it does place some requirements on how the model is written. For more information, see the Aside below.

Finally, there are some unit tests. In this case these are nothing more than evaluations of the Model at specific q values. When a Plugin Model is compiled any unit tests are evaluated so that SasView can check it is using the Model correctly.

```
tests = [[{}, 0.001, 1009.98],
         [{}, 0.150141, 0.175645],
         [{}, 0.442528, 0.0213957]]
```

Aside: Models can be much more sophisticated than this example, especially if computing unusual shapes, magnetism and/or orientations. Models can also be pure Python, as in this example, or Python calling external C subroutines, or Python with embedded C code.

For further information, see
https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/plugin.html.

Now let us repurpose this model.

SasView already has the *guinier* Model that computes

$$I(q)=scale\,.\exp\left[\frac{-q^2 R_g^2}{3}\right]+background$$

but let us suppose we want this function expressed in terms of the spherical radius, *R*

$$I(q)=scale\,.\exp\left[\frac{-q^2 R^2}{5}\right]+background$$

We can edit *example4.py* as follows:

Change the Model description, in particular the equation.

```
# new_guinier model
# Note: model title and parameter table are inserted automatically
r"""
Definition
----------

The scattering intensity I(q) is calculated as

.. math::
    I(q) = \text{scale} \cdot exp [\frac {-q^{2}  R^{2}} {5}] + \text{background}

References
----------

#. Guinier, A.

Authorship and Verification
----------------------------

* **Author:**  **Date:**
* **Last Modified by:**
* **Last Reviewed by:**
"""
```

Import the exponentiation function from *numpy*.

```
from numpy import inf, errstate, exp
```

Change the Model name and title.

NB: At the time of writing, the name of the Plugin Model and the name of the Plugin Model file must match; here, *example4.* This restriction may be removed at a later date.

```
name = "example4"
title = """Calculates a Guinier function with the particle radius."""
description = """
"""
category = "shape-independent"
```

Define a new set of parameters.

```
#             ["name", "units", default, [lower, upper], "type","description"],
parameters = [
              ["sld", "1e-6/Ang^2", 1.0, [-inf, inf], "", "SLD of particle"],
              ["sld_solvent", "1e-6/Ang^2", 6.0, [-inf, inf], "", "SLD of solvent"],
              ["radius", "Ang", 50.0, [0, inf], "", "Radius of particle"],
             ]
```

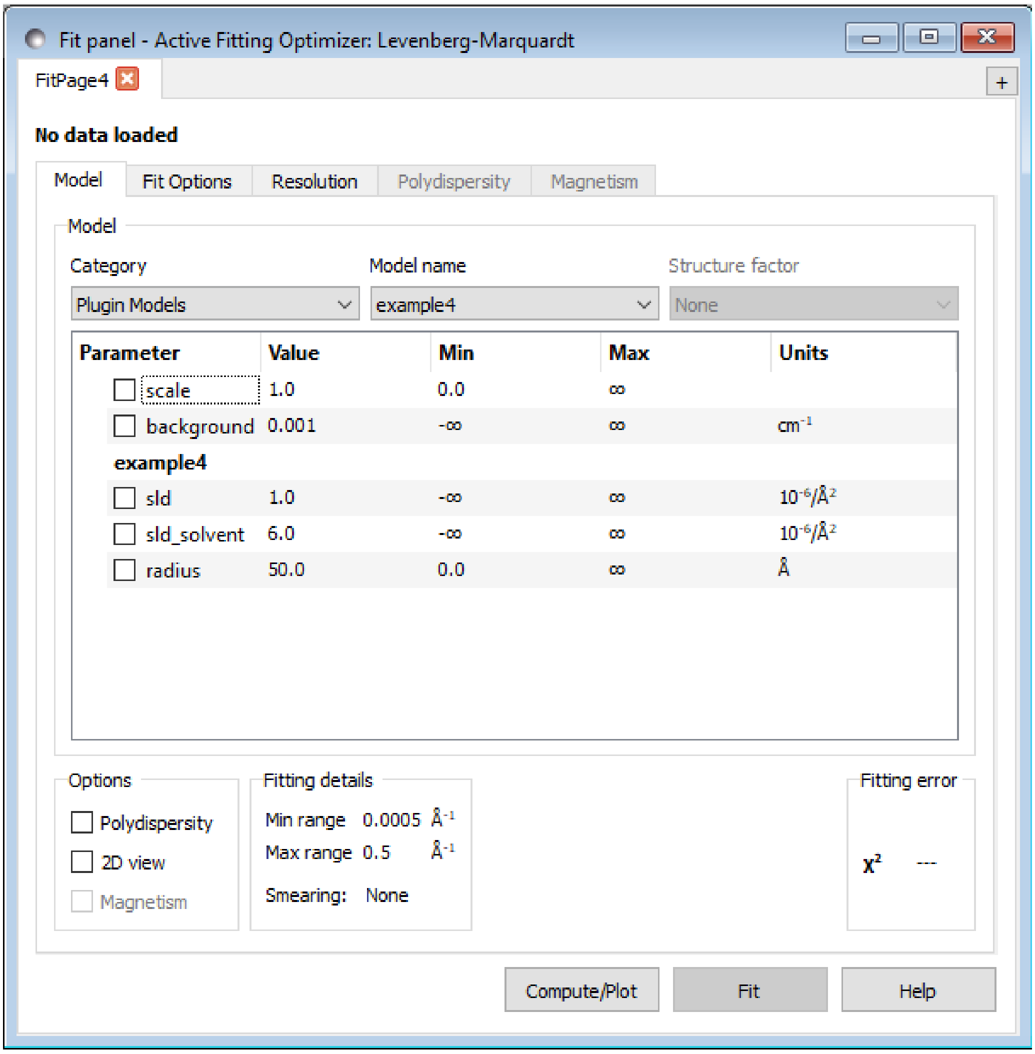
Change the Model function definition.

```
def Iq(q, sld, sld_solvent, radius):
    """
    1D calculation of a Guinier sphere
    """
    with errstate(divide='ignore'):
       inten = (sld - sld_solvent)**2.0 * exp(-q * q * radius * radius / 5.0)
       return inten
Iq.vectorized = True
```

And for expediency, comment out the unit tests for now, even if it is bad practice!!!

```
#Tests = [[{}, 0.001, 1009.98],
#          [{}, 0.150141, 0.175645],
#          [{}, 0.442528, 0.0213957]]
```

Save the plugin file *example4.py.* Then open a new *FitPage* in SasView and load the plugin *example4.* The Model should display like the example below.



No data loaded

Model

Category: Plugin Models

Model name: example4

Structure factor: None

Parameter	**Value**	**Min**	**Max**	**Units**
☐ scale	1.0	0.0	∞	
☐ background	0.001	-∞	∞	cm^{-1}
example4				
☐ sld	1.0	-∞	∞	10^{-6}/Å^2
☐ sld_solvent	6.0	-∞	∞	10^{-6}/Å^2
☐ radius	50.0	0.0	∞	Å

Options

☐ Polydispersity

☐ 2D view

☐ Magnetism

Fitting details

Min range 0.0005 Å^{-1}

Max range 0.5 Å^{-1}

Smearing: None

Fitting error

χ^2 ---

If *example4* is not visible in the Model name dropdown box then SasView encountered a problem compiling the Model. The most likely reason for this will be a syntax error, and the simplest way to find out is to load the Model in the *Model Editor*; **Fitting** > **Edit Custom Model** > Load Plugin...

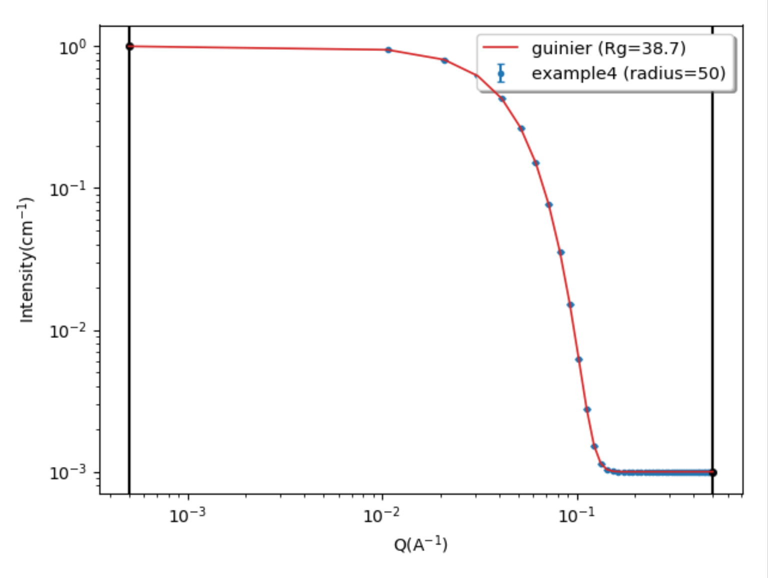
Fix the error, click Save, and try reloading the Plugin Model.

Tip: If you copy and paste code from this document you may encounter this error when you try and load the Plugin Model:

```
SyntaxError: invalid character in identifier
```

Sometimes, depending on your operating system and the text editor you are using, non-text characters, like minus signs, for example, paste in a form that looks right visually but which is not recognised by Python (the character 'encoding' is wrong). If this happens, just try deleting these characters in the code and replacing them with the 'true' character typed from the keyboard.

If the repurposed Model is working as expected then it should be computing a scattering curve the same as that from the base library *guinier* Model if the radius-of-gyration, *Rg*, in that is matched to the *radius* in *example4* (recall that R_g^2 = (3/5) radius^2). And indeed it does if the contrast term introduced into *example4* but not present in *guinier* evaluates to 1 (so simply make one of the *sld* parameters equal to 1 and the other equal to 0).



Example 5 – Reparameterising An Existing Model

A typical use case here is where there is an existing Model that could be of use, but one or more of the parameters it is optimising are not the parameters you require. Whilst you could use the knowledge you have already gained from this tutorial to construct a new Model, SasView has a relatively simple method of allowing you to reparameterise an existing Model without actually editing it.

Consider the existing *sphere* Model in the base library. This is currently parameterised in terms of the spherical radius, the parameter *radius*.

But let us suppose we wish to do the reverse of **Example 4** and reparameterise this Model in terms of the spherical radius-of-gyration, *Rg*. As encountered in the previous example, the necessary transformation is R_g^2 = (3/5) radius^2 or, inverting this, radius = $(5/3)^{1/2}$ R_g.

Launch a text editor of your choice. If it is a 'language-sensitive' editor sensitive to Python, all the better (for example, the SasView *(Custom) Model Editor*, *Notepad++* or *PyCharm*).

Create a new file and enter the following code (the example below has been generated in *Notepad++*; for an explanation of the colour coding see **Example 4**):

Tip: If you use the (*Custom) Model Editor* it will not be necessary to restart SasView in order to pick up the modified Model.

```
from numpy import inf
from sasmodels.core import reparameterize

parameters = [
    # name, units, default, [min, max], type, description
    ["Rg", "Ang", 50, [0, inf], "volume", "Sphere Rg"],
]

translation = """
    radius = sqrt(5.0/3.0)*Rg
    """

model_info = reparameterize('sphere', parameters, translation, __file__)
```

Now save the file to your plugins folder, say, as *example5.py.*

The existing model being reparameterised is declared in the last line: '*sphere*'.

The *Rg* parameter declaration replaces that of the existing *radius* parameter in the *sphere* Model when *example5* is loaded as a Plugin Model.

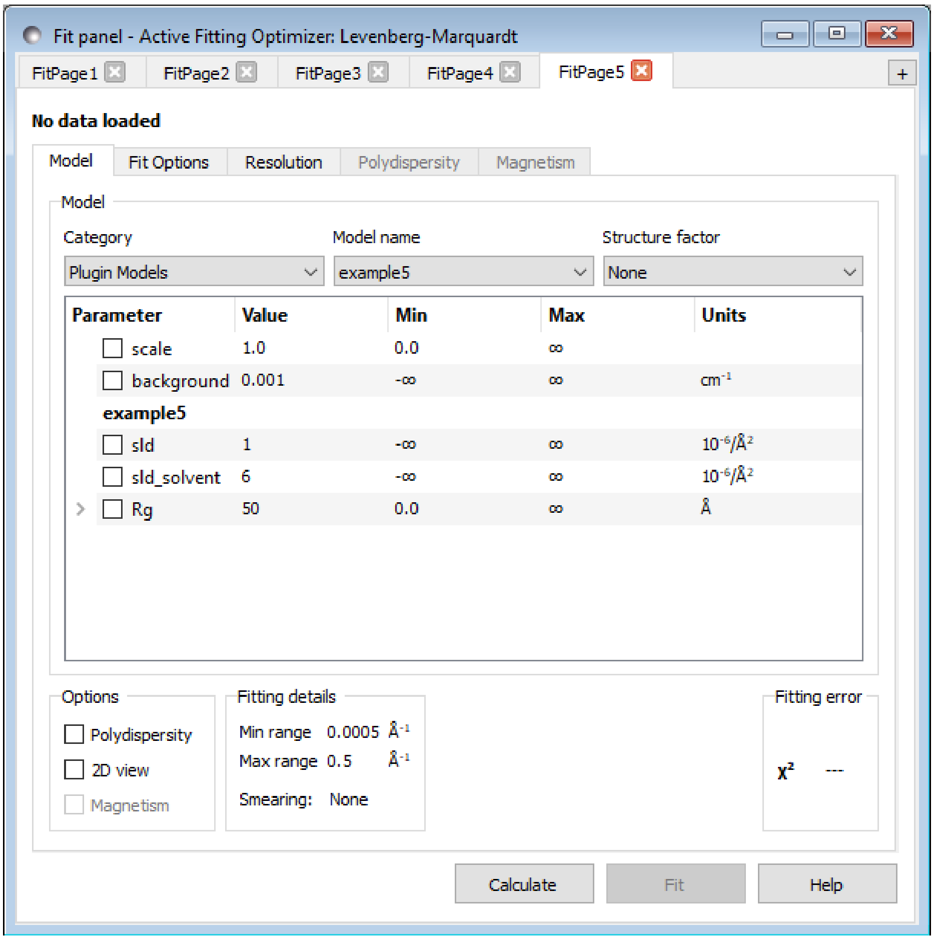
And the translation declaration specifies how the existing parameter is computed from the newly-declared parameter.

Tip: The code fragment above is sufficiently generic that only the portions in yellow highlight would need to change if reparameterising another Model.

At the time of writing there are some Models whose parameters cannot be fully translated.

Examples include: *core_multi_shell*, *onion*, *spherical_sld*, *rpa*, and *unified_power_rg*. These Models have multiple instances of some parameters, where the number of instances are variable and determined by a control parameter, such as the number of shells or a calculation case number. Such parameters cannot presently be translated.

Open a new *FitPage* in SasView and load the plugin *example5.* The Model should display like the example below.



No data loaded

Model

Category: Plugin Models
Model name: example5
Structure factor: None

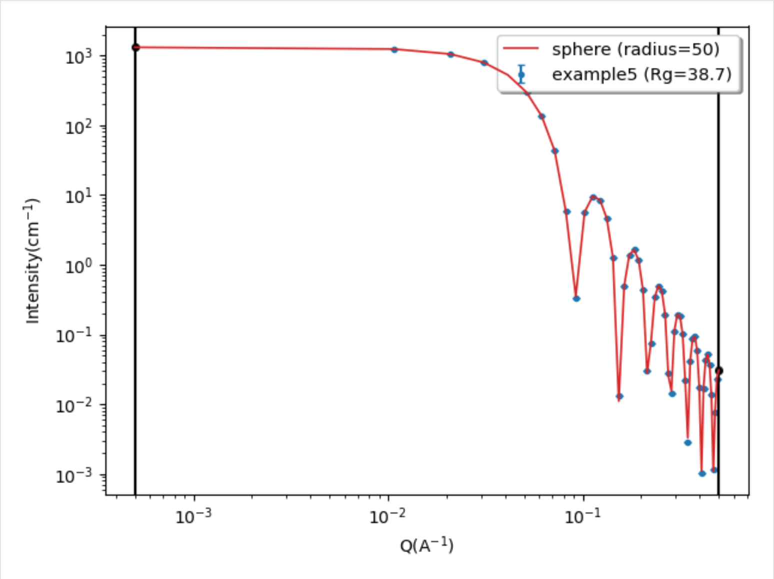
Parameter	Value	Min	Max	Units
☐ scale	1.0	0.0	∞	
☐ background	0.001	-∞	∞	cm^{-1}
example5				
☐ sld	1	-∞	∞	$10^{-6}/\text{Å}^{2}$
☐ sld_solvent	6	-∞	∞	$10^{-6}/\text{Å}^{2}$
☐ Rg	50	0.0	∞	Å

Options
☐ Polydispersity
☐ 2D view
☐ Magnetism

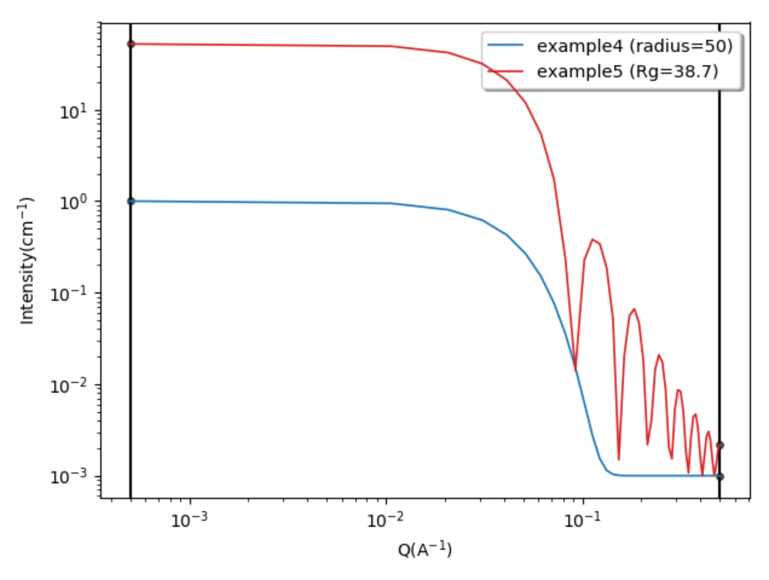
Fitting details
Min range 0.0005 Å^{-1}
Max range 0.5 Å^{-1}
Smearing: None

Fitting error
χ^2 ---

If the reparameterised Model is working as expected then it should be computing a scattering curve the same as that from the base library *sphere* Model if the radius-of-gyration, *Rg*, in *example5* is matched to the *radius* in *sphere*. And indeed it does if the contrast terms (ie, the *sld* parameters) are the same in both Models.



Aside: You might think that Models *example4* and *example5* would also give broadly similar scattering curves, at low-Q values at least, if computed with the same *sld* values. But they do not.

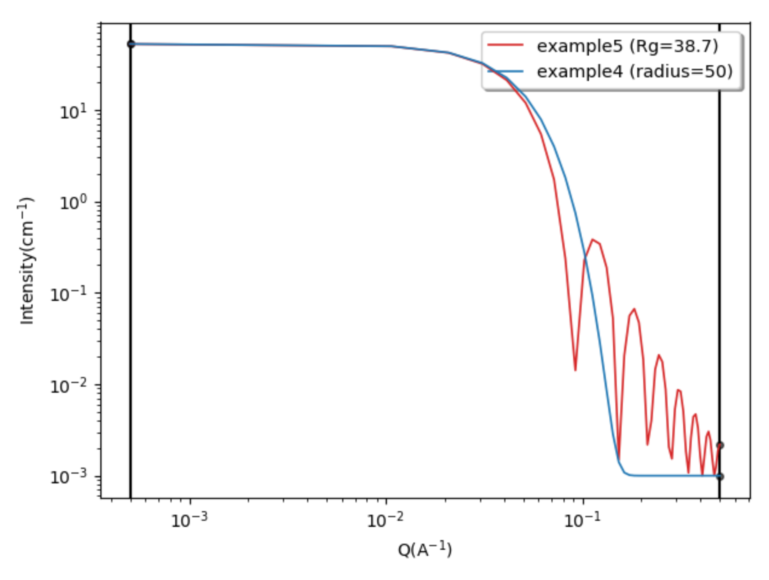


The reason is that the *guinier* model has no volume (*V*) normalisation applied to it. This model is a limiting case and is volume independent meaning a volume normalising term is inappropriate. See: https://www.sasview.org/docs/user/models/guinier.html

radius = 50 Å = $50\text{x}10^{-8}$ cm; (*sld* – *sld_solvent*) = $1\text{x}10^{-6}$ Å^{-2} = $1\text{x}10^{10}$ cm^{-2}

So the missing prefactor is $V(\Delta sld)^2 = (4/3).\pi.(50\text{x}10^{-8})^3.(1\text{x}10^{10})^2 = 52.35$ cm^{-1}

If this is applied to *example4* as *scale*=52.35 then the Models converge.



Example 6 – Getting Creative

The obvious use case here is where no existing Model, or combination of Models, are of use. But another use case is where you want to allow a Model to be combined with a Structure Factor from the dropdown on the FitPage, as opposed to creating multiple Plugin Models as shown in Examples 1 and 2.

Whatever your intentions, a good place to start is the SasView documentation page at: https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/plugin.html

As we have seen in **Example 4** there is a defined ordered structure to a Model:

- **the Model Description**
  - written in ReST/LaTeX
  - enclosed between the lines `r"""` and `"""`

- **any Function Imports**

```
from numpy import ...
```

- **the Model Definition**

```
name = "model_name"
title = """short one line explanation of the model"""
description = """
slightly longer explanation of the model; can be across
multiple lines like this
"""
category = "shape-independent or shape:catergory"
```

- **the Model Parameters**
  - Remember *scale* and *background* are added automatically to all models
  - Try and re-use existing parameter names where possible!
  - Parameter names should also follow mathematical convention; so *radius_core* and not *core_radius*, for example

```
parameters = [
["param1", "units", default_value, [min_value, max_value],
"parameter_type", "short description of parameter1"],
["param2", "units", default_value, [min_value, max_value],
"parameter_type", "short description of parameter2"],
]
```

If the scattering depends on orientation then it is also necessary to include the angles theta, phi and psi at the end of the parameter table. See the section on 'Oriented Shapes' in the SasView plugin documentation at: https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/plugin.html

- **the Model Function**

- There are various ways this might be defined.
- In a **pure Python Model**, add a code block that computes the required function; for example, in *correlation_function.py*:

```
def Iq(q, lorentz_scale, porod_scale, cor_length,
porod_exp, lorentz_exp):
    """
    1D calculation of the Correlation length model
    """
    with errstate(divide='ignore'):
        porod = porod_scale / q**porod_exp
        lorentz = lorentz_scale / (1.0 + (q *
        cor_length)**lorentz_exp)
    inten = porod + lorentz
    return inten
```

- In a **Python Model with Embedded C** code, add the C code between ””” quotes; for example, in *guinier.py*:

```
Iq = """
    double exponent = fabs(rg)*rg*q*q/3.0;
    double value = exp(-exponent);
    return value;
"""
```

- In a **C Model**, a calling Python model of the same name must specify the necessary C routines; for example in *capped_cylinder.py*:

```
source = ["lib/polevl.c", "lib/sas_J1.c", "lib/gauss76.c",
"capped_cylinder.c"]
```

- In this last example computation also requires three SasView library functions, see the plugin documentation for further details.

Tips:

- C is faster than Python!
- Only C Models can run on a GPU. But also remember not to set `opencl = False`!
- Only C Models currently support the use of orientational distributions and magnetism.
- Only C Models currently support the direct calculation of $\langle F(Q)^2 \rangle$ or $\langle F(Q) \rangle^2$, for example, as is used in the beta-decoupling approximation to the Structure Factor. Remember to set `have_Fq = True`!

If the scattering depends on orientation then it is also necessary to compute Iqabc(qa, qb, qc, param1, param2,…) or, if there is rotational symmetry about the c axis, Iqac(qab, qc, param1, param2,…), etc. See the section on ‘Oriented Shapes’ in the SasView plugin documentation at:
https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/plugin.html

- **the Model Computation**
    - There are various ways to control the behaviour of the Model, see the plugin documentation for details.

```
opencl = False                 # optional: defaults to False
                               # set to True if the model
                               # should be able to use GPUs

single = True                  # optional: defaults to True
                               # sets the precision to be used
                               # on GPUs; True = single,
                               # False = double

structure_factor = False       # optional: defaults to False
                               # set to True if the model is
                               # an S(Q) rather than a P(Q)

have_Fq = True                 # set to True if the model also
                               # has an F(Q) function defined
                               # (which you may also need to
                               # supply); currently only
                               # used by the beta
                               # approximation calculation

Iq.vectorized = True           # set to True if the Iq
                               # calculation can compute with
                               # a vector q (ie, all values at
                               # once)
                               # This is being deprecated
```

    - If a shape Model can be used with a Structure Factor it is possible to specify what characteristic length(s) of the shape can be tied to the effective interaction distance of the S(Q) function. An *unconstrained* option is also added automatically. For example, in *capped_cylinder.py*:

```
radius_effective_modes = [
    "equivalent cylinder excluded volume",
    "equivalent volume sphere",
    "radius",
    "half length",
    "half total length",
 ]
```

    - The calculations for these modes are found in *capped_cylinder.c*.

Note that an effective radius function needs to be defined in order for the *structure_factor_mode* drop-down to be enabled in the Fit Page.

  - (Deprecated and best not used!) An alternative way of defining the effective radius is as follows:

    ```
    def ER(radius):
        radius = …
        return radius
    ```

  - And if two parameters are correlated but it is known that one must be larger than the other a constraint can be added as follows:
  - `valid = "param1 >= param2"`

- **the Model Normalisation**
  - For the calculated intensities from a shape model to be on an absolute scale they must be correctly normalised to the volume of the shape. This is accomplished by defining the *form_volume*. The I(Q) calculation should then use *form_volume* as its scale factor.
  - Not defining *form_volume* is equivalent to `form_volume = 1`.

    If *form_volume* is not provided then the volume normalisation must be incorporated in the model calculation, however, z-average parameters will be returned instead of number-average parameters.

  - In a pure Python Model or Python Model with Embedded C:

    ```
    def form_volume(param1, param2,...):
        volumeis = …
        return volumeis
    ```

  - In a C Model:

    ```
    static double
    form_volume(double param1, param2,...)
    {
     // the calculation
     const double volumeis = ...
     return volumeis;
    }
    ```

  - Hollow shapes, that is, shapes where the volume fraction of material resides in the shell rather than the whole volume of the shape, should also define an equivalent *shell_volume*.
  - The I(Q) calculation should then use *shell_volume* as its scale factor.
  - But if *shell_volume* is not defined the shape is assumed to be homogeneous.

  - As a structure factor calculation needs the volume fraction of the filled shapes for its calculation, the volume fraction parameter in a hollow shape model normally needs to be scaled prior to calling the structure factor. This can be accomplished by defining the volume ratio, *VR*:

```
def VR(param1, param2,...):
    ratiois = form_volume/shell_volume
    # or some variation of this formula
    return ratiois
```

- **the Model Unit Tests**
    - Unit tests are a way of getting SasView to check that it is running your Model correctly. In essence, you provide a Python dictionary of parameter input values and the corresponding q-intensity output values.

      ```
      tests = [
         [{'param1': value1, 'param2' : value2, ...},
          [q1, intensity1], [q2, intensity2]],
         ]
      ```

    - The example above represents quite a simple set of tests, but much more testing functionality is available, see the plugin documentation for details. Alternatively have a look at the library model *sphere.py* !
    - If testing a Model that also operates in 2D, simply replace 1D input q values by their (q_x, q_y) tuples.

Further Information

For further information, please consult

http://www.sasview.org

https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/fitting_help.html#adding-your-own-models

https://www.sasview.org/docs/user/qtgui/Perspectives/Fitting/plugin.html

http://marketplace.sasview.org/

or email

help@sasview.org